A Top Layer Design Approach to Complex Real-Time Software

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Science

Jair Jehuda

June 3, 1997

Acknowledgments

First and foremost, the author wishes to thank his trusted supervisors: Daniel M. Berry, Amos Israeli, Dror Zernik, the Ph.D. Committee, and other Godsent messengers, for their expert advice in safely guiding this ship through uncharted seas to new horizons. When I started out, I already had ten years of practical experience in the development of large and complex real-time systems, but it was Dror who first introduced me to the theoretical world of real-time computing and who whet my appetite to attempt to bridge the gap between the two worlds. As top-layer architecture and algorithms started taking shape, it was Amos who taught me the art of analyzing their performance and formulating them in writing. But more than anything else, it was Dan's wisdom, humor, and encouragement, which kept me steadily on course from beginning to end, down to the finish line.

Due to the inherent complexity of the issues, this thesis had to grapple with several diverse problems which required specialized expertice. Consultants from a variety of fields came to my rescue: Gilad Koren (BIU) on real-time scheduling, Adam Shwartz (Technion) on markov decision processes, and Yishai Weisman (Technion) on extreme-value statistics. There were also several landmark discussions which helped focus and inspire: with Dennis Shasha (NYU) and Jack Stankovic (UMASS) who helped put my work into perspective; and with Jacob Katznelson (Technion), Avi Mendelson (Technion), Ben Reiser (HU), and Neeraj Suri (Applied Systems) who propelled me forward by asking the right questions. All of the above deserve special thanks for reading preliminary drafts of my papers and for their many helpful comments and suggestions.

Needless to say, adequate funding is a prerequisite for any research project, primarily when one has a family of ten to sustain. My sincere gratitude to Rafael for granting me a crucial three year fellowship, as well as to the Technion for the generous supplementary scholarship so sorely needed for survival.

Last, but not least, immense gratitude to my wonderful wife Dina, who nourished me and encouraged me throughout; and to my eight darling children: Yishai, Noam, Benia, Elchanan, Yonatan, Daniel, Eden, and Rachel, for without their considerate cooperation, none of this would have been possible.

Abstract

In this thesis we introduce a *top-layer* design approach to *complex* real-time software, so called because it is designed to produce and support *platform-independent* software for extremely complex system requirements, e.g., concurrent, best-effort hard real-time systems running on shared-memory multi-processor platforms.

The keys to our approach are simple, yet flexible platform and job-oriented program models that enable decomposing larger, generally highly complex, scheduling problems into several smaller, generally simpler, scheduling problems, called *jobs*. Internal job scheduling problems are independently resolved offline and encapsulated in a platform-independent manner, creating reusable real-time software objects. A hard real-time *job time-sharing* scheme is then used to incorporate these jobs into programs without having to deal with the scheduling details within them. An additive schedulablility criteria suffices for determining whether a given set of jobs can reliably time-share a given processor, thereby greatly reducing the complexity of *load-balancing* over available platform resources.

To best utilize any given platform, each job may also support several *modes* of operation, each mode requiring a different measure of resources and contributing a different reward to an application-specific system value function. Job mode alternatives, required mode resources, and contributed rewards may also be state-dependent, varying with computational circumstances. An appropriate software *meta-control* algorithm must therefore occasionally recompute the job modes which maximize the system value per time unit for evolving runtime circumstances without overloading the system resources, i.e. without jeopardizing any hard deadlines within any of the jobs on any of the platform processors. This multi-mode job-oriented program model thus introduces a novel notion of *best-effort* meta-control in a hard real-time context. The same model can also be used to enable several *concurrent* hard real-time applications to reliably share appropriate multi-processor platforms.

Appropriate algorithms have been devised, several policy making issues have been addressed, and all of the above have been incorporating within a comprehensive top-layer architecture. A realistically complex musical real-time application has also been implemented to demonstrate the feasibility of our approach. Results are very promising, clearly indicating that platform-independent, best-effort hard real-time software is a viable and realistic goal for many products, with great potential for significantly reducing real-time software development and maintenance costs.

Contents

List of Figures

List of Tables

Glossary of Symbols

C_i	computation time for task i	24
T_i	task period for task i	24
$u(\tau)$	processor utilization for task set τ	24
C	set of CPUs (processors)	37
c	CPU id	37
β_c	computational capacity for CPU c	37
J	set of jobs	38
J_c	set (cluster) of jobs allocated to CPU c	38
j	job id	38
τ_j	task set for job j	38
δ_j	scheduling policy for job j	38
f_j	minimal processing speed for job j	38
b_j	bandwidth for job j	38
M_j	set of modes for job j	39
m_j	current mode for job j	39
r_j	reward for job j	39
v	total system value	39
s	system state	41
d_j	earliest deadline for job j	52
o_j	worst-case scheduling overhead cycles per task in job j	58
f_j	slowest feasible dedicated processor speed for job j	58
n_i	number of execution cycles for task i	58
u_j	bandwidth utilization for job j	61
α_j	time-sharing overhead factor (b_j/f_j) for job j	61
njobs	number of application jobs	77
ncpus	number of platform processors	77
u	total capacity utilization	78
γ	a clustering policy, e.g. FFD	78
γ^*	an optimal clustering policy	78

$\bar{u}$	minimal utilization which guarantees clusterability	89
$\mathcal{J}$	the current application (jobs) state	100
$\mathcal{C}$	the current platform (cpus) state	100
ω	a mode vector identifying a selected mode for each job $j \in J$	100
$\omega[j]$	identifying the $m_j \in M_j$ mode selected for job j	100
$v(\omega)$	the system value provided by mode vector ω	100
$b(\omega)$	the total bandwidth required by mode vector ω	100
$u(\omega)$	the total capacity utilization for mode vector ω	100
Ω	the set of all possible mode vectors over $\mathcal{J}$	100
ω_0	the minimal bandwidth mode vector in Ω	100
$\Omega_{\mathbf{c}}$	the set of clusterable mode vectors in Ω	101
$\mathbf{c}(\omega)$	indicator whether $\omega \in \Omega_{\mathbf{c}}$	101
$\mathbf{c}$	a cluster vector identifying a job cluster for each job in J	101
$\mathbf{c}[j]$	a cluster vector identifying the job cluster for job $j \in J$	101
$v^*(\Omega, b)$	max system value for all $\omega \in \Omega$ such that $b(\omega) \leq b$	101
ω^*	an optimal mode vector for current system state	102
v^*	the optimal system value for current system state	102
π	a given meta-control policy	103
ψ_π	value performance for meta-control policy π	103
$\bar{v}$	a v^* upper bound	103
$\bar{\psi}_\pi$	pessimistic ψ_π lower bound for meta-control policy π	103
$\mathbf{J}_j$	job list of (band, reward, mode) tuples for job j	104
$\mathbf{J}_j^k$	the kth tuple in job list $\mathbf{J}_j$	104
$\mathsf{ntuples}(\mathbf{J}_j)$	number of Job j tuples	104
nmodes	total number of system tuples	104
β	total platform capacity	104
q	QDP bandwidth quantum	106
ω_π	mode vector selection by policy π	106
$\mathbf{c}_\pi$	clustering solution for policy π	106
ρ	mean mode vector load (total capacity utilization)	117
Ψ_π	r.v. of sampled ψ_π for random system states	117
$\Omega_{\mathbf{c}}$	the Ω subset which is clusterable	139
$\mathbf{c}(\omega)$	boolean function indicating if ω is clusterable	139
$P(\mathbf{c}\|u)$	clusterability of mode vectors in Ω with utilization u	139
λ	a policy-making policy	161
S	the set of anticipated system states	161
T_p	a common periodic task period in ATLAS	210
T_f	time til ATLAS FIFO queue underflows	210

Abbreviations and Acronyms

Chapter 1

Introduction

As hardware costs transcend and software requirements grow, the unwavering high cost of software development and maintenance has become a very powerful driving force behind the development of "write once, run anywhere" technologies such as Ada, Postscript, HTML, SQL servers, OpenGL, Java, CORBA and many others. Such open system technologies essentially enable the programmer to concentrate on the *top* application layer only, with lower firmware and hardware layers requiring only minimal attention, thereby significantly cutting costs by substantially simplifying the software development and maintenance process. But perhaps even more important, the software produced by these technologies is essentially *platform-independent,* enabling it to run on a wide variety of available platforms, to be readily ported to new platforms when old ones get obsolete, and to consistently perform better as newer platforms grow faster and more powerful.

Unfortunately, the impact of these rapidly developing platform-independent software technologies has hardly been felt in the area of real-time computing. It is widely assumed that real-time software and platform-independence are inherently mutually exclusive. After all, *real-time* systems are those which must interact with their environment in a timely manner. Untimely interactions in *hard* real-time systems, e.g. those found in defense, transportation, aerospace, energy, and health care, can be very hazardous and costly, and are therefore not to be tolerated. The real-time software within such systems is typically responsible for carrying out a substantial set of ongoing activities which must be appropriately scheduled to complete their functions within carefully crafted time intervals. The reliable scheduling of these perpetual activities clearly requires complete *a priori* knowledge of the available run-time resources as well as the computational and other resource requirements for each activity. Typical scheduling solutions are therefore thought to be inherently platform-dependent.

There are, of course, exceptions to this rule. In very *simple* real-time systems, such as those in which all activities are purely periodic, completely independent of each other, and all running on a single processor, relatively simple schedulability criteria and dynamic real-time scheduling policies can be applied uniformly to all candidate platforms. The hard real-time applications which we wish to address in this thesis are typically much too *complex* to be scheduled in a platform-independent manner, thereby causing them to be prohibitively expensive to develop and maintain.

This thesis hopes to convince the reader that appropriate models, control algorithms, and software architectures can indeed produce platform-independent software even for hard real-time systems which are quite complex. To this end, we have developed a novel notion of *best-effort* hard real-time systems, i.e. systems which have hard real-time requirements and are also required to dynamically adapt their behavior to best exploit whatever run-time resources are currently available to them. As we will show, when running on shared-memory, multi-processor platforms, the requirements of such best-effort hard real-time systems are so complex that there are no known methods for addressing these complexities even when the runtime platform is *a priori* known and fixed. The *top-layer* design approach introduced by this thesis shows how such complexities can realistically be addressed, and in a *platform-independent* manner. Furthermore, we show that hard real-time applications which have adopted the approach can then run *concurrently* and reliably on appropriate multi-processor platforms. We thus demonstrate that platform-independent complex real-time systems are not only possible, but that a top-layer design approach can also help resolve complexities which would otherwise be computationally intractable.

The scope of our research includes

- developing appropriate platform and aggregate job-oriented program models, including a novel notion of platform-independent *best-effort* meta-control in a hard real-time context, which can also be applied to concurrent real-time applications,
- devising practical job load-balancing, hard real-time job time-sharing, and best-effort meta-control algorithms,
- addressing several policy making issues, e.g. how to select an optimal

suite of algorithms for a specific application,

- incorporating the above within a comprehensive top-layer architecture,
- demonstrating the feasibility of the approach by using it to implement a realistically complex real-time application which is portable and best-effort, and
- producing a comparative tradeoff analysis of the top-layer approach with existing approaches regarding cost, overhead, the ability to meet scheduling constraints and more.

Results are very promising, clearly indicating that concurrent, platform-independent, best-effort hard real-time software is a viable and realistic goal for many systems, with great potential for significantly reducing real-time software development and maintenance costs. True, in its current form, the top-layer approach developed here does *not* serve as a solution for *all* real-time applications. Nevertheless, by showing how it can be applied, as is, to a very challenging class of real-time systems, and by suggesting how current models and algorithms can be extended to embrace a wider class of applications, we hope to pave the way for future developments which will ultimately increase the portability and adaptability of real-time software.

The purpose of this chapter is to provide the necessary background for understanding what this thesis is all about and why the thesis is structured the way it is. First, we establish the setting by introducing the complex real-time problem domain in Section 1.1, and listing primary motivations in Section 1.2. Section 1.3 then briefly introduces the top-layer approach, describing the assumed platform and program models, outlining the software control elements required by these models, and using a few brief examples to illustrate how the top-layer architecture might be applied differently to different applications. This is followed by Section 1.4 which introduces ATLAS, a realistically complex real-time application which has been successfully implemented using the top-layer approach. Section 1.5 then whets the appetite a bit more by relying on our ATLAS experience to suggest that future top-layer enhancements might also better accommodate non-deterministic elements in run-time platforms and environments. Section 1.6 concludes with a description of the thesis structure, outlining the purpose of each chapter and providing a synopsis of what each chapter has to offer.

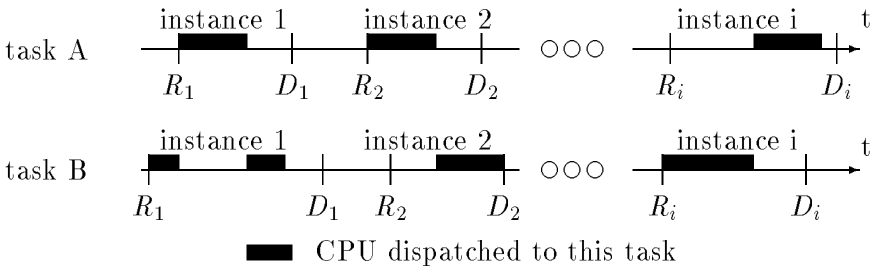

Figure 1.1: EDF Task Scheduling

1.1 Complex Real-Time Systems

The problem domain of this thesis is a class of real-time applications which we call *complex real-time applications* because *the online decisions that they require are of such complexity that optimal decisions are intractable.* More specifically, we wish to address real-time applications which have hard (tight) timing constraints, difficult-to-schedule task characteristics, and dynamic, state-dependent, task sets and execution times. We further require that these applications reliably run, as is, on any of several platforms, each platform having a different number of shared memory processors, and each processor possibly running at a different machine cycle rate. When provided with various alternate mode of behavior, we require that these applications should also online adapt their behavior to best accommodate each platform and each application state. Finally we require that they should also be able to reliably run concurrently alongside additional complex real-time applications on any of these designated platforms. As we later show, *conventional task-oriented design approaches are not equipped to deal with such complexities even in a sub-optimal manner.*

The following subsections introduce real-time systems, describe the scheduling and meta-control decisions required by such systems, and better define the targeted complexities and assumed limitations.

Real-Time Systems

Real-time systems are those that must interact with their run-time environments in a timely fashion. Such systems are typically modeled as running *perpetually*, comprising a well-defined set of software modules called *tasks*, many of these tasks being invoked periodically or sporadically an infinite number of times to carry out their particular functions. As depicted in Figure 1.1, each task invocation is called an *instance*. Each time a task instance i is invoked, it must usually complete its function within a well-defined time interval, the beginning of that interval being referred to as the task *release* time, R_i, the end of that interval being referred to as the task *deadline*, D_i.

Scheduling and Schedulability

Intervals for several concurrent tasks will often overlap, so an appropriate real-time task *scheduler* must determine when each of the invoked tasks instances will be provided with the resources needed for task execution. The computational resources required by each task are usually provided in units of time, indicating how much time it would take for the current run-time processor to complete the task. Given the task computation times, release times, and deadlines, the real-time scheduler attempts to dispatch the processor to the various task instances in a manner which enables them to meet their deadlines. One effective *policy* for doing this is to always dispatch the task which has the earliest deadline. Such a policy is referred to as EDF (earliest deadline first) scheduling and Figure 1.1 demonstrates how such a policy would successfully dispatch a shared processor to the sample tasks, A and B without missing any deadlines.

Given a set of tasks, a schedule is *feasible* if all tasks can be scheduled such that all timing constraints and resource requirements are met. Major scheduling issues include whether a feasible schedule *exists* for a given task set, whether such a schedule will be *found* by a given scheduling policy, and whether these questions can be resolved by a reasonable *schedulability* test or criteria.

Periodic and Sporadic

As depicted in Figure 1.1, task release times, deadlines, and computation times can be quite irregular, but this complicates scheduling and schedulability. *Periodic* tasks, on the other hand, greatly simplify scheduling, so real-time systems are modeled as sets of *periodic* tasks whenever possible. *Periodic* tasks are invoked repeatedly at fixed time intervals, i.e., letting T denote the fixed time period between instances, then $R_{i+1} = R_i + T$ for such periodic tasks. Periodic task instances must typically complete before their next instance release, so we also assume that $D_i = R_i + T$ for such tasks, unless otherwise specified.

Real-time systems, however, must almost always react to aperiodic events as well, e.g., external interrupts and internal signals. Such events invoke the execution of event-driven tasks which are referred to as *sporadic* tasks. Sporadic task release times are therefore not *a priori known*, but they can still be analyzed for schedulability, if they are invoked at rates which never exceed a specified upper-bound. Thus we require that all sporadic tasks in our applications have a known lower-bound, T, such that $\forall i : R_{i+1} - R_i \geq T$.

Hard and Soft, Values and Gain

Deadlines may be hard or soft, whereupon tasks are referred to as hard or soft in accordance with the nature of their deadlines. *Hard* deadlines are those for which the system fails to meet its requirements if a deadline is not met, while missed *soft* deadlines merely cause system degradation. The measure of degradation is often defined by a time-dependent *value* function attached to each soft task, which represents the contribution of each task instance to an aggregate *system value* as a function of *when* the task instance completes relative to its deadline. A common scheduling criteria for such soft tasks is to maximize the aggregate system value per time unit, henceforth referred to as the system *gain*.

Mandatory and Optional

Another important distinction is made between mandatory (critical) and optional (non-mandatory) hard tasks. When dealing with *mandatory* hard tasks, *all* instances must be executed and all must complete on time. The instances of *optional* hard tasks must execute only if they can complete on time. As in soft tasks, a *value* must be associated with each optional task instance to serve as a scheduling criteria for preferring one optional instance over another. Thus a common scheduling criteria for hard tasks is that all instances of the mandatory tasks always be executed and completed on time, while instances of optional tasks are chosen to maximize the system gain, and they, too, are executed only when they can complete on time. Throughout this thesis we will assume that all hard tasks are mandatory unless otherwise specified.

Processor Utilization

When seeking a cost-effective processor for a given set of perpetual real-time tasks, the primary criteria is the schedulability of hard tasks and an acceptable system gain for the soft tasks. A common *secondary* criterion is to maximize the processor *utilization*. When dealing with a set of purely periodic tasks, τ, each task $i \in \tau$ having a task period T_i, and a fixed computation time, C_i, for all instances of task i, then the processor utilization, $u(\tau)$, is readily computed by

$$u(\tau) = \sum_{i \in \tau} \frac{C_i}{T_i}. \tag{1.1}$$

We later find that schedulability and processor utilization are strongly related. Thus, for example, clearly, not all instances of τ are schedulable if the processor utilization exceeds one. As hardware costs transcend and software costs grow, the significance of achieving a high processor utilization is often questionable. In many applications, however, platform costs, volume, weight, and power consumption is clearly limited, whereupon the ability to maximize the utilization of the available platform resources can be a major issue.

Real-Time Applications

We define a *real-time* application as one comprised of arbitrary mixtures of perpetual periodic and sporadic tasks. In a *hard* real-time application, all tasks are *hard*, while in a *soft* real-time application, all tasks are *soft*. The real-time systems we wish to address will usually contain a mixture of both hard *and* soft tasks. A major challenge for the successful implementation of such real-time systems is finding a cost-effective platform and a practical algorithm for *scheduling* these tasks in a manner that *always* guarantees the hard deadlines while ensuring that the soft tasks contribute an *acceptable* system gain.

Practical real-time scheduling algorithms have been studied extensively in recent decades, and it is generally well-known whether a given scheduling policy can reliably schedule a given task set on a given run-time platform. The easiest case to handle is when all tasks are periodic, preemptive, independent, and with fixed computation times per task. *Preemptive* tasks are such that their execution can be interrupted at any point of execution to execute a higher priority task. *Independent* task are those which have no precedence constraints and lack shared resources which require synchronized access, so that no task can be *blocked* by another task and therefore each task can be scheduled without considering how other tasks are scheduled. As we show in Chapter 2, such task sets can be reliably scheduled on a single processor using any of several simple online scheduling policies, subject to well known schedulability constraints. Thus, for example, an EDF (earliest deadline first) scheduling policy will always work in such systems if the worst-case processor utilization does not exceed 100 percent, while an RM (rate monotonic, i.e. highest rate first) scheduling policy, will always succeed if the worst-case processor utilization does not exceed 69 percent (and typically higher).

Optimal Hard Scheduling

In a hard real-time context, a given scheduling policy is considered *optimal* if it always finds a feasible schedule if such a schedule exists. It is well known that when using conventional priority-driven scheduling, then EDF is optimal for the above systems when task priorities can be dynamically altered in accordance with the scheduling policy [Der74]. RM scheduling, on the other hand, is optimal for the above systems when the scheduling policy must determine a fixed-priority for each task [LL73]. As already mentioned, both policies have very simple utilization-based schedulability criteria. Both policies are also *practical* in the sense that online scheduling overheads are generally proportional to the number of tasks. RM scheduling overheads are lower, thanks to the fixed priorities, while EDF-scheduled processor utilizations are generally higher. No wonder that both policies are quite popular in hard real-time systems.

Best-Effort Soft Scheduling

In a soft real-time context, a given scheduling policy might be considered *optimal* if it maximizes the *expected* system gain. When dealing with independent sporadic tasks and fixed task priorities, allocating higher priorities to tasks with the higher value densities is known to be optimal [Whi92], the task value *density* being equal to the task value divided by the average task execution time. Due to their sporadic nature, task release times may vary from one run to another, so there may very well be a different allocation of priorities which would provide a higher system gain for specific runs. Nevertheless, given that these task release times are random, the above density-based priorities should maximize the mean system gain over a significant enough number of runs. Scheduling policies which maximize the *expected* gain are generally referred to as *best-effort* policies.

Speculative Soft Scheduling

When dealing with non-independent soft tasks, e.g. tasks which require synchronized access to shared resources, density-based priorities might not maximize the expected gain due to the blockage of higher density tasks by lower density tasks. A less greedy policy, on the other hand, might achieve a higher gain by only scheduling task instances when all of their required resources are available, thereby reducing blockage. Heuristic policies which have been shown to often be effective, but which lack proved best-effort qualities, will henceforth be referred to as *speculative* policies.

Complex Real-Time Applications

Practical scheduling and schedulability criteria get more complicated when dealing with multi-processor platforms, with dynamic task sets, and with more complicated task set characteristics, such as those having both hard and soft deadlines, those having both periodic and sporadic tasks, those having varying computational needs, those having tasks with non-preemptive code segments, and those having inter-task dependencies such as those caused by precedence-constraints, communicating tasks, and shared resources. First of all, such systems require a *load-balancing* scheme for allocating tasks or task instances to each of the available processors without overloading any of these processors. Once allocated, appropriate scheduling policies must schedule the tasks on each of the processors while taking into consideration the unknown release times of the sporadic tasks as well as non-preemptive code segments and inter-task dependencies. Moreover, dynamic task sets require effective and practical load-balancing and scheduling algorithms which can be carried out online with reasonably low overheads.

The targeted problem domain for this thesis includes applications with all of the above forms of complexity, but with the following restrictions, all for simplicity's sake:

1. We assume that tasks are computationally intensive and that the platform processor capacities are the only limiting factors when carrying out task load-balancing. Memory and I/O resources are therefore

currently assumed to be infinite, even though extensions to multi-dimensional resource limitations appear straight-forward.

2. We assume that the dynamic task sets and task computational requirements are *state*-dependent, in the sense that the application has an *a priori* known set of states, and each state has a known fixed set of tasks with known worst-case computational requirements per task. This would enable us to resolve most load-balancing and scheduling issues offline for all possible states if the number of states were reasonably small.

3. We generally assume that system states vary only *occasionally*, i.e. at significantly low rates, so that load-balancing and task scheduling for each new state can be carried out knowing that the new state will persist long enough to be considered perpetual. We later show, however, when and how this assumption can be relaxed.

4. We assume that for each possible system state, there exists an *a priori* known uniprocessor scheduling solution, i.e. a scheduling policy and computational capacity such that the policy can be shown to adequately schedule the required task set for that state when these tasks share a single processor with the given capacity. To accommodate all possible system states, all we have to do is provide a single processor with the worst-case computational capacity. A single processor with the worst-case capacity will usually not be available on the targeted multi-processor run-time platform, but this constraint guarantees, at least, that the task characteristics for each system state are such that the schedulability of any given task *subset* on any given processor can be determined offline, as well.

For the sake of brevity, real-time applications belonging to the above multi-processor problem domain, will henceforth be referred to simply as *complex* real-time applications. As we later show in Chapter 2, the paradigms serving current commercial and research real-time platforms cannot offer practical solutions for such applications. Real-time extensions to commercial operating systems, e.g. Real-time UNIX [FGG+91] and Real-time MACH [TNR90], generally cater to best-effort *average* performance and are therefore inappropriate for *hard* real-time. Scheduling in the distributed MARS platform [DRSK89] is static, and therefore inappropriate for *dynamic* task sets. Spring [SR89] strives for a mode flexible combination of off-line and on-line scheduling techniques, but it, too, must rely on static load-balancing and scheduling when dealing with mandatory hard tasks. The object-oriented MARUTI system [LTCA89] focuses on fault-tolerance and on-line FCFS (first-come-first-serve) guarantees using a non-preemptive, calendar-based, scheduling scheme, so it cannot accommodate dynamic task *characteristics* or best-effort adaptation to dynamic run-time circumstances. The distributed ARTS kernel [TM89] uses online, RM-based, preemptive priority-driven scheduling, which can more readily cater to dynamic task sets and dynamic, value-oriented, load-balancing, but task priorities are static and therefore inappropriate for dynamic task characteristics. RM schedulability gets much more complicated when dealing with complex task characteristics, so ARTS can be impractical for dynamic task sets when they have complex task characteristics. The object-oriented CHAOS kernel [GS89] focuses primarily on efficient tailoring of the real-time kernel to arbitrary platforms, thereby enhancing the portability of its applications, but it, too, uses various forms of FCFS guarantee-oriented scheduling which cannot readily accommodate complex task characteristics. Thus, when dealing with hard real-time tasks, all of the existing platforms use static scheduling or online-FCFS guarantee scheduling, neither of which can support *best-effort adaptation* to dynamic run-time circumstances.

The inability of these platforms to cater to our complex problem domain is not surprising. It is well known [MD78] that *optimal* hard real-time scheduling on a multi-processor platform is NP-complete even for the much simpler case of purely periodic and independent tasks. As is shown in the survey of Chapter 2, when dealing with *complex* real-time systems with *hard* deadlines, even non-optimal policies can become impractical for online use since these cannot provide *a priori* guarantees that *all* hard deadlines will always be met. A *portable* solution must therefore somehow devise *offline* solutions to cater to all possible system states and all members of a family of platforms. But, as we soon point out, the growth of possible platform configurations and system states is usually exponential, so that such an approach would be intractable, as well. Thus, the development of a *top-layer* design approach for such systems is a formidable challenge.

1.2 Motivation

The quest for less complicated, more flexible, and less expensive real-time software is the primary motivation for seeking a platform-independent, *top-layer* design approach for complex hard real-time systems. Another primary motivation for portability is that it paves the way to two additional features of our top-layer approach: the ability to support concurrent real-time applications and best-effort adaptation in a hard real-time context.

In many real-time applications, customizing the application to run on a new platform and getting it to run concurrently with additional applications can be a relatively simple matter. This, however, is rare with *complex* real-time applications, and even more rare when these applications have *hard* tasks. Before we show how these goals are obtained, we must first understand why traditional bottom-up approaches are inherently limited in these respects.

Traditional Bottom-Up Approaches

As previously mentioned, choosing a *cost-effective* run-time platform is a major challenge for complex real-time applications. A very powerful platform, i.e. one which would result in a platform utilization which is much lower than what can be obtained by an optimal scheduler, might indeed minimize the task coding effort and simplify scheduling, but it also results in higher *production* costs. A less powerful platform might require more optimized code and a more sophisticated scheduling scheme, thereby incurring higher *development* costs. Traditional real-time design approaches work *bottom-up* from the platform hardware to the top-layer application, working from known times for hardware instructions to guaranteed worst-case upper bounds on the executions of system and library routines invoked by the application tasks, to guaranteed worst-case predictions on the timing behavior of the tasks themselves. As already mentioned, when dealing with a complex real-time application, finding an adequate load-balancing and real-time scheduling solution for a given platform can be a very computationally intensive process, and it must also consider every possible system state.

A bottom-up approach essentially forces us to commit to a particular platform very early in the system design when precise task requirements and characteristics, e.g. worst-case execution times, can still be quite vague. Thus, the run-time platform is essentially chosen before we can adequately evaluate its cost-effectiveness. Alter the nature, number, and capacity of the available platform processors and we must once again use computationally intensive algorithms to devise new load-balancing and scheduling solutions.

Portability and Real-Time Software Costs

The inherent platform-dependence of bottom-up real-time software has very serious implications on real-time software development and maintenance costs.

Major difficulties arise when poor initial estimates or evolving system requirements result in finding the targeted specifications unrealistic or the chosen platform to be inappropriate. For example, such difficulties are common place in cutting-edge projects in which little can be learned from past project experiences, or in which available processing resources are often limited by volume, weight, and power consumption. Such difficulties generally have a severe impact on project scheduling and budget, and this impact is often magnified by greater task scheduling complexities because new scheduling solutions may require radical code changes. Note that once the chosen platform has been found inadequate, porting a complex application even to a platform with an *equal* number of *faster* processors might require new scheduling solutions. The new schedulers may be required either because the original cyclic executive solution assumes minimal execution times for certain activities and these assumptions no longer hold, or because the original scheduling solution is no longer possible because the improvements in the original worst-case execution time estimates are not evenly distributed among the tasks. Porting a complex real-time application to a more cost-effective platform, e.g. to a smaller number of more powerful processors, will require a re-distribution of tasks among processors as well as new scheduling solutions for each processor.

Platform-independent software is a valuable asset even when the targeted platform is indeed appropriate. Application debugging and testing are formidable tasks for many real-time systems, and portable applications can be developed and tested on platforms with the most effective tools available and then ported to the production platforms, which are often bereft of such tools. Portability facilitates software development, eases software maintenance, encourages software reuse, and increases application longevity by enabling easy porting to new platforms when initial platforms become obsolete. Thus any design method that produces software that is more portable is worth its weight in gold.

Concurrent Real-Time Applications

Another primary motivation is the growing need for platform-independent real-time applications which can run concurrently and reliably along-side additional platform-independent real-time applications. As platform-independent Java-coded multimedia applets grow more prevalent, we require a design method that will enable arbitrary sets of applets to reliably share a given run-time platform if the resources of that platform are adequate. In a conventional approach to real-time scheduling, each set of applets and each platform poses a different scheduling problem which would have to be resolved online to determine whether platform resources are adequate and if yes, what the scheduling solution should be. As already stated, for complex real-time applications, the online resolution of such problems would be computationally intractable. A major top-layer design objective is to enable us to resolve each application's scheduling needs independently and offline, and to do so in a manner which will support an online time-sharing mechanism which will consider the real-time deadlines of each application and enable us to quickly evaluate whether available platform resources are adequate.

Best-Effort Adaptation

The design of portable real-time applications makes way to a new challenge which was only mildly relevant for platform-dependent applications in the past. When designed to run on a given platform, an application will be generally be designed *a priori* to best exploit the available resources of that platform. When designed to run on any of several platforms, an application aught to be designed with inherent flexibility so that it can adapt to best exploit the available resources on each run-time platform. Even when designed for a specific platform, an application with state-dependent characteristics aught to best adapt to each state. To distinguish between this and best-effort scheduling, this will be referred to as *best-effort adaptation.* To this end, we introduce the notion of a set of application *modes* of operation to be chosen from, each mode known to contribute to some notion of a system gain. The current application mode must then be selected for each platform and for each state in a manner that will maximize the system gain without overloading the platform resources, e.g. without missing any hard deadlines. Different modes may employ different algorithms, different task sets, and different task requirements, e.g., different task periods. Such multiple-mode applications will henceforth be referred to as *best-effort* applications, and we will generally assume a large number of modes to best accommodate a large variety of states and platforms.

Best-effort adaptation can also be useful in a *fault-tolerant* context. When one of the platform processors fail, best-effort applications will automatically and gracefully degrade their performance, not necessarily by eliminating lower priority tasks, but rather by altering the task modes so that their required resources will not overload the reduced set of available resources.

Unlike the above-mentioned notion of best-effort *scheduling* which can only be applied to *soft* real-time applications, the suggested notion of best-effort adaptation can be applied to *hard* real-time applications as well, since each application mode can employ a different set of tasks with hard deadlines. The essential difference is that values, here, are used to indicate mode contributions, whereas in best-effort scheduling, values are used to describe the contribution of completing each task deadline.

Once developed, a best-effort application's performance will only improve with age. As we latter demonstrate, a missile simulator will automatically provide more precise results, a wireless base station will automatically provide better service, a virtual reality program will automatically provide a more realistic immersion, a flight control system will automatically provide better navigation paths and smoother landings, and a multimedia program will automatically provide higher quality audio and video when ported to a more powerful platform.

However, maximizing the system gain in a *hard* real-time system implies that the schedulability of *all* mode alternatives must be tested for each system state on each platform. As previously mentioned, online schedulability testing for a multiprocessor platform is prohibitive even for a small set of alternatives. When dealing with large-scale applications, each application subsystem may have several mode alternatives, yielding an exponential number of mode combinations. When catering to concurrent applications, the number of system-wide mode combinations can grow exponentially large even when each application has only a few mode alternatives. For large numbers of mode combinations, online schedulability testing for all of them can be impractical even when dealing with purely periodic and independent tasks on a single processor. Thus, conventional approaches cannot support best-effort hard real-time applications when task characteristics are complex, when the number of mode alternatives is significant, or when running on a multiprocessor platform. As we will show, the top-layer approach presented here is designed to make best-effort applications a viable alternative even for concurrent, hard, and complex real-time systems.

1.3 A Top-Layer Approach

Our goal, therefore, is to produce real-time software that can readily port from one shared-memory multi-processor platform to another without altering the program and without having to devise new scheduling solutions for each new platform. The run-time platform can therefore be finalized much later in the development cycle when all the necessary data are available.

The keys to our approach are simple yet flexible platform and aggregate job-oriented program models that should enable decomposing larger, generally highly complex, scheduling problems into several smaller, generally simpler, scheduling problems that can be resolved independently and offline, and then encapsulated in a platform-independent manner. Each of these smaller scheduling problems are referred to as a *job*, and software components developed using the approach can be treated as reusable *real-time objects* that should be readily incorporated into new programs without having to deal with scheduling details within them. Well designed jobs should facilitate near-optimal platform utilizations. Evolving system requirements could then be readily resolved with the related jobs without effecting the other jobs in the system. Schedulability is at the core of all real-time systems, so this modularity in the scheduling domain should greatly reduce the complexity of load-balancing and the making of best-effort mode selection decisions.

1.3.1 The Platform Model

We will generally assume that all targeted platforms belong to the same *family* in the sense that task execution times on any given processor within the family are assumed to be equal to the number of machine cycles needed to complete the task, divided by a processor characteristic called the processor *capacity*. This notion of capacity essentially reflects the processing *speed* of that processor. All platforms within a family are assumed to differ only in the number and capacity of the shared-memory processors in each platform.

When all processors share a common machine language and the execution of each machine instruction requires an equal number of machine cycles for all processors, then the processor *capacity* is simply equal to the inverse of the machine cycle time and therefore prescribed in MHz units. For such platform families we seek *binary-level portability*, whereby top-layer applications should run on an arbitrary number of shared-memory processors with arbitrary processor capacities without modification. This restricted notion of a platform family is later extended in Chapter 3 to embrace *source-level portability*, i.e. different processors and different machine languages, as well. Once the family of processors has been established, each platform is fully characterized by a set of processor capacities.

Note that the primary obstacle to portable real-time applications is the complexity of resolving scheduling in a uniform manner for platforms with different task execution times and different resource capacities. Even when assuming portability over a specific family of processors, different processor numbers and capacities can produce an exponential growth of different scheduling problems. When dealing with hard complex real-time applications, resolving scheduling and schedulability for all possible platform configurations would therefore be intractable. Thus the development of portable real-time applications is a formidable challenge even when limited to specific processor families.

1.3.2 The Static Top-Layer Program Model

In the *static* program model, the software application is modeled as a set of real-time subsystems, called *jobs*, each job comprising of any number of periodic and sporadic real-time tasks. As previously described in Section 1.1, the computational requirements of each task are typically provided in time units and for a specific processor. Our platform model assumes a specific family of processors, but the processor capacity (speed) may vary, so the task computational requirements are therefore specified, instead, in machine cycles. Real-time scheduling is independently resolved offline for each job by initially assuming that the job task set runs on a dedicated processor, by selecting an appropriate conventional policy for scheduling the job's internal tasks, and by computing the minimal processor capacity, i.e. the minimal rate of machine cycles needed, to guarantee all *hard* deadlines and the minimal *soft* value contributions required by that job. One job, for example, could be employing a cyclic executive policy (see Section 2.1.7) to schedule its tasks, and will be known to satisfy its temporal requirements when run on a processor with a machine cycle rate of 2 MHz or more, while another job might use a rate-monotonic scheduling policy (see Section 2.1.4) for its tasks, and its minimal machine cycle rate might be 10 MHz.

Thus, the top-layer program model can be applied to arbitrary job sets, provided that each job consists of a task set that has a known scheduling solution for an arbitrary known processor capacity. The real-time task characteristics and scheduling solutions are encapsulated within the jobs by making each job personally responsible for its internal scheduling, e.g., by including a scheduling task or thread for facilitating the chosen scheduling policy. The tasks of a job could even be completely serialized at compile time to eliminate the need for this run-time scheduling task.

Job Time-Sharing

The load-balancing granularity in our model is the program *job*, i.e., it is understood that all the tasks in a given job will always run together on a single processor, always share the same memory, and otherwise compete with other jobs for resources offered by the shared platform. Once a satisfactory scheduling algorithm is determined offline and implemented for each job, then the normally dynamic, on-line scheduling of all the tasks in the system, i.e., of all the tasks in all the jobs, can be reduced to that of the dynamic, on-line scheduling of only the jobs. Once a job is scheduled, its internal offline-determined scheduling algorithm determines which of its tasks will run within its allocated running slots. An appropriate real-time job *time-sharing* scheme can then enable several jobs to share a given platform if it can guarantee that each job will be dispatched in a timely manner and allocated the required rate of machine cycles such that the job's internal scheduling policy will always reliably schedule the job's tasks. An *overloaded* processor is one which has been allocated a job set for which such guarantees cannot be provided by the job time-sharing mechanism. The jobs running on a given platform may belong to a single application or may belong to several concurrent applications, so the same scheme can be used to time-share any number of real-time applications.

The Job Bandwidth

To accommodate the above time-sharing scheme, the minimal rate of machine cycles required for each job must be appropriately augmented to accommodate the worst-case job time-sharing overheads. This augmented rate of machine cycles is referred to as the job *bandwidth*, a term used to denote a periodic allocation of machine cycles as in the bandwidth-preservation techniques of [LSS87, SSL89]. We then require that the job time-sharing scheme on each processor must reliably guarantee scheduling for an arbitrary job set if the sum of job bandwidths does not exceed the given processor capacity. As we show in Chapter 3, different time-sharing schemes can be used for different classes of complex real-time applications, with EDF job time-sharing being quite effective when dealing with computationally intensive tasks and relatively relaxed task frequencies.

Another interesting job time-sharing feature is that *minimal* bandwidth allocations are guaranteed for *all* jobs in the system so that jobs with low priority tasks never starve when running together with jobs containing higher priority soft tasks that would otherwise fully load the processor. The proposed EDF job time-sharing scheme may appear trivial, but we know of no previous attempt to thus resolve in this way complex scheduling problems of this nature.

Black-Box Jobs

We also require that the job time-sharing mechanism need know only the job bandwidths and the current earliest deadline for each job. This enables top-layer jobs to be treated as *black-box* real-time software objects in the sense that they can be reliably load-balanced and time-shared without having to divulge their internal task characteristics and scheduling policies. This contrasts with conventional approaches where the real-time platform can only cater to the needs of *white-box* applications, i.e. applications which reveal all tasks and a significant set of task characteristics. Each job sharing a given processor can therefore employ a totally different scheduling policy, e.g. CE, EDF, or RM with PCP. Major job design goals should include minimizing the job bandwidths and maximizing the job *bandwidth utilizations*, i.e. the rate of machine cycles actually consumed by the job tasks divided by the job bandwidth. If these goals are achieved, time-shared, real-time scheduling can be near optimal. Job bandwidths and utilizations should therefore be carefully considered when decomposing the software application into jobs and when selecting scheduling policies for each job.

Job Load-Balancing

Schedulability is at the core of all real-time systems. Therefore, this black-box modularity in scheduling domain greatly reduces the complexity of load-balancing when the job is the basic load-balancing entity. As previously mentioned, arbitrary job sets may share a given processor if the sum of job bandwidths does not exceed the processor capacity. Thus the allocation of jobs to processors on any given shared-memory multi-processor platform is essentially a generalization of the classic binpacking problem, and the processor capacities should generally be great enough to contain several jobs. Binpacking is strictly NP-complete, but it has several practical algorithms, e.g. first-fit-decreasing, which we have appropriately adopted to carry out job load-balancing in our model. A given platform can therefore serve any application if the job bandwidths of that application can be partitioned over the processors of that platform without overloading any of the processors.

This static program model is portable in the sense that job load-balancing is readily carried out on any platform without having to alter the scheduling algorithms being internally used by each job. This scheme is also readily applied to systems running several concurrent real-time programs, in which a request to run a program is accepted if the subsequent aggregate job set can be load-balanced over the available processors based solely on the job bandwidths within the programs. Thus job bandwidths and appropriate algorithms for real-time job time-sharing and load-balancing provide a simple mechanism for determining whether arbitrary top-layer programs can reliably run on arbitrary platforms.

1.3.3 The Dynamic Top-Layer Program Model

The dynamic, top-layer program model assumes that system requirements and available resources can be represented by well-defined platform and program *states* that vary at relatively low rates. The *platform state* reflects the available platform resources and it is altered when a processor fails or when there is a *hot* (on-the-fly) reconfiguration of hardware. The *program* (or workload) state is altered when a user request or other run-time factors require changes in the current task mix and/or affect the task execution times and how quickly these tasks must respond. Such changes clearly alter the minimal processing bandwidths required by each job to satisfy its task needs, so the current allocation of jobs to processors may no longer be efficient or valid.

To address the needs of such systems, the static top-layer program model is extended by letting each job support one or more alternate *modes* of operation, sometimes referred to as job *versions*, each mode encapsulating an arbitrary set of tasks and an appropriate internal scheduling policy for satisfying each task's deadlines when it is provided with a minimal *bandwidth* of machine execution cycles. Each job mode is therefore fully characterized by the required bandwidth along with a *reward* representing the aggregate value contributed by the internal tasks when provided with the given bandwidth. These job mode bandwidths are already determined offline along with the internal scheduling policies for each job mode, so they must be provided as tables or functions of the *a priori* known program states. Thus in a cellular base station application, (see Section 1.3.3) the bandwidth for handling a telephone session should vary as a function of the current session mode, e.g. the chosen sample rate and error-correction method, as well as the state of the session, e.g. whether connecting, speaking, or faxing. Job mode rewards, also, may be offline determined as functions of the program state, e.g. a higher session sample rate will contribute very little when connecting and in other states the contribution will depend on the spectral nature of the information being transferred. *Suspendable* jobs are also supported by the dynamic program model. Jobs that can be suspended have a zero-bandwidth, zero-reward suspend mode to indicate that possibility.

In the model, the aggregate system gain must be adequately expressed as a function of the rewards provided by the current job modes. In some cases, the system gain function might be additive, i.e. equal to the sum of current job mode rewards. Best-effort accommodation for each state is accomplished by the online selection of job modes that maximize the system gain for that state without overloading the available processors. Thus, all job mode bandwidths and rewards must be reevaluated with each altered state, whereupon they must be considered in the re-selection of job modes and the re-allocation of jobs to processors. Each job may have an arbitrary number of states, and the application state is a combination of current job states, so the number of system states can be exponential. For such cases, the offline resolution of load-balancing and scheduling at the task level would be intractable for all possible systems states even when dealing with relatively simple task sets.

Online Meta-Control Decisions

The process of choosing between available software adaptations, e.g. alternate job modes, is referred to as software *meta-control* because it essentially determines which tasks must be scheduled, what their time constraints are, and possibly which scheduling policy should subsequently be used. In the dynamic top-layer program model, online software meta-control decisions require best-effort job mode selection and job load-balancing, which is shown to be equivalent to a composite binpacking and multiple-choice knapsacking problem, which is NP-hard. Online meta-control decisions must be carried out with each program or platform state transition. Therefore, minimal and reasonable decision-making overhead is crucial. To minimize the response time for such changes, we have developed two families of practical approximation algorithms, QDP and G^2, for making near-optimal best-effort meta-control decisions with reasonable time complexity, typically providing 95% performance in fractions of a second for substantial numbers of jobs, modes, and processors.

In our top-layer program model, task characteristics, constraints, and scheduling policies are encapsulated within each job mode. Thus each job must also implement an online mechanism for providing the required bandwidth for each available mode as a function of the current system state. We assume that extensive offline schedulability analysis for each job mode can produce efficient online mechanisms for providing these state-dependent job mode bandwidths, and a similar online mechanism must be provided for the state-dependent job mode rewards as well. The system must therefore respond to each state transition by querying each job for its current mode alternatives, mode bandwidths, and mode rewards so that it can make a best-effort meta-control decision.

Meta-Control Decision Epochs

As already described in Section 1.1, we assume in our model that the rate of altered system and platform states is reasonably low so that task sets remain stable enough for conventional scheduling policies to be applied to them over a significant time interval. Furthermore, we assume that software configuration adaptations, i.e. transitions from one job mode to another, can either be carried out instantly or be done quickly enough that these adaptation overheads can be accommodated. As we soon show by example, many applications can be designed to meet these limitations, and we also believe that these limitations will eventually be relaxed by appropriate extensions to our current meta-control algorithms. We also assume that meta-controller decision-making processes can be accommodated without disrupting essential real-time processes. One way of avoiding disruption is to dictate that all meta-controller decisions take effect at discrete points in time called *decision epochs*, and to ensure that task deadlines are adequately synchronized with these epochs so that new tasks can be introduced and old tasks can be flushed simultaneously.

A Virtual Reality Example

Consider, for example, a virtual reality program required to maximize the overall realism produced by simulating and displaying realistic behavior of a dynamic set of visible entities. Each entity can be treated as a job with several modes, each mode using a different simulation update rate. The realism contributed by each entity's mode could be a factor of the entity's state, e.g. the nature of its motion as well as observed size and visibility, so the reward attributed to each entity must consider how the update rate for that entity will contribute to the overall realism in its current state. Higher update rates might even imply different simulation models requiring a different set of tasks and constraints, so that the task set serving each entity mode may be different. Once the entity modes have been determined, it is essential that all hard deadlines associated with these tasks be met. To facilitate seamless meta-controller adaptations, we might decide that adaptations may only occur at epochs that are at times that are multiples of 100 milliseconds. This decision implies that all other update rate alternatives should be multiples of 10 Hertz as well. This would enable the system to instantaneously switch job modes, i.e., update rates and simulation models, at any given adaptation epoch, since none of the tasks belonging to the replaced simulation models should be executing at those points in time. Another interesting possibility might be to carry out the adaptations in stages, whereby all transitions to lower bandwidth modes are carried out first, asynchronously, at their first possible opportunities, and all other transitions can then follow in a carefully crafted sequence which never overloads the system. The existence of such a sequence would have to be ascertained as part of the meta-controller decision-making process. Section 3.3.3 elaborates on this virtual reality example.

A Cellular Base Station Example

There are, however, many examples in which meta-controller decision epochs and task timing constraints need not be synchronized. In such a case, the need for predetermined decision epochs and instantaneous transitions are not necessary. In a cellular phone base station, for example, a rising number of subscribers or a faulty processor would cause the base station to gradually curtail a variety of voice enhancement and noise-cancellation processes so that a uniform highest overall service level is maintained for the current number of subscribers and processors. A base station has several seconds to decide whether or not it can respond to a new subscriber, and how it can best adapt to the new work load. This provides plenty of time for gradually phasing out old tasks and phasing in new tasks, without risking any missed hard deadlines in tasks serving current subscribers. This example illustrates how even significant meta-control adaptation overheads might be handled when the system is allowed an adequate response time to changes in its run-time requirements. Section 3.3.4 elaborates on this base station example.

Policy-Making

The issue of *when* to make meta-controller decisions, *which* meta-control policy to use, which jobs should be considered, and *how* to best carry out the necessary adaptations is generally referred to here as *policy making*. As illustrated by the above examples, policy making is very application dependent. We therefore touch upon only certain aspects of policy-making in this thesis, such as how to measure and estimate meta-control policy performance for a given application and how an adaptive *follow-the-leader* scheme can be used to obtain online optimal policy making when run-time circumstances, i.e. the distribution of system states, is not known *a priori*.

1.4 The ATLAS Experiment

Once the algorithms have been devised, it is necessary to formally derive their computational complexity to determine whether they have a chance of behaving as hoped. However, this formal modeling is not enough. Theoretical estimates are only orders of magnitude and it is not known what the multiplicative constants are. It is necessary to build and run a system designed according to the top-down principles, making use of these algorithms, in order to verify the effectiveness of the approach. To this end we have devised a scalable, best-effort real-time system with a realistic mix of hard and soft deadlines, with very frequent meta-control decision epochs, capable of porting unaltered to any of several accessible platforms, while meeting all hard deadlines and maximizing the system gain for each platform.

The system developed is a music synthesizing system called ATLAS, an acronym for *a top-layer audio synthesizer*. All digital synthesizers produce digital samples that must be output via appropriate digital-to-analog interfaces at a predetermined sample rate. As depicted in Figure 1.2, the digital samples produced by ATLAS are all software computed and fed into a hardware queue (FIFO) in real-time, so the digital samples for each note must

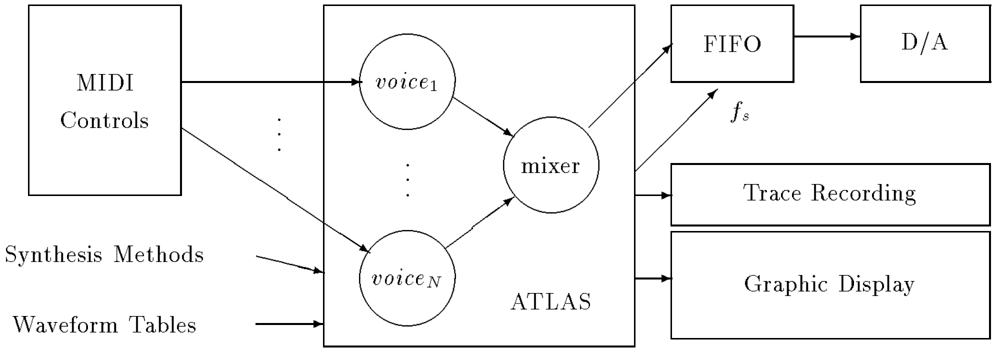

Figure 1.2: The ATLAS Application

be generated and mixed at rates that sustain the flow of output samples via the hardware queue. The notes generated by ATLAS are dictated by a standard online MIDI[1] input driven by several electronic instruments, each instrument producing sequences of requests to start or stop the playing of one or more concurrent notes, e.g. when playing a chord. Each ATLAS note that has to be generated is referred to as a *voice*, with higher pitched voices requiring higher sample generation rates. Thus the internal application state is a function of the number of concurrent MIDI-dictated voices, the nature of the instruments, and the voice pitches, and ATLAS is required to best adapt to each state on any of the targeted runtime platforms.

To accommodate different platforms and dynamic system states, ATLAS provides several synthesis modes (methods) of varying quality for generating the digital samples for each instrument. Each mode is characterized by the computational bandwidth requirements of the mode and the signal-to-noise ratio produced by the mode. As illustrated in Figure 1.3, each synthesis mode may require any number of periodic tasks. An *FM* (frequency modulation) synthesis mode, for example, requires an **osc** oscillator task for generating a periodic stream of digital samples which produce a voice waveform with an appropriate pitch, another **fm** task for generating values which frequency modulate the oscillation pitch, and a third **env** task for generat-

[1]MIDI is an acronym for the standard Musical Instrument Digital Interface Specification 1.0, which was first established by the International MIDI Association in North Hollywood, California in 1983.

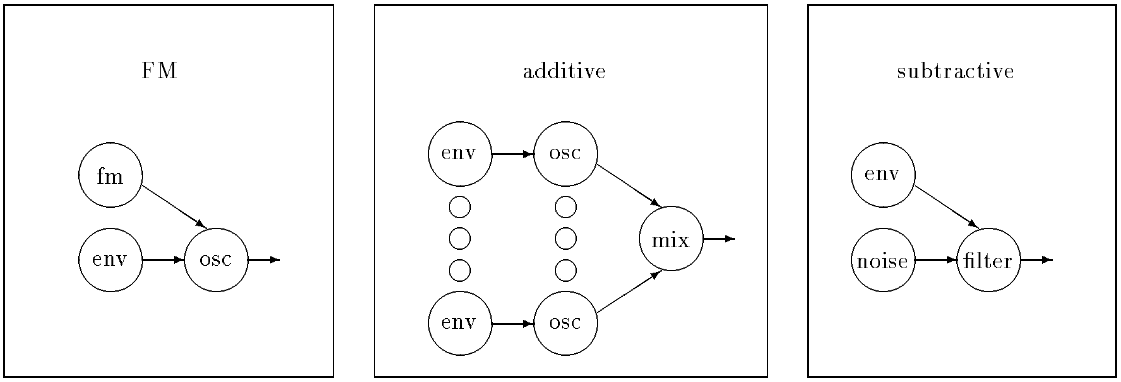

Figure 1.3: Typical Musical Synthesis Methods

ing oscillation amplitude values to obtain an appropriate power envelope for each note. An *additive* synthesis method may require any number of oscillator and envelope tasks along with a mixer task to mix their outputs. A *subtractive* synthesis method will require a noise generation task, a digital filtering task to obtain a specific frequency spectrum, and an envelop task to modulate the amplitude. These tasks are periodic in the sense that they are periodically invoked to process and generate sample *packets* containing several samples at a time. These packets can then serve as inputs to other synthesis tasks, including the final mixer task of Figure 1.2 that mixes the samples of all concurrent notes before feeding its output packets into the final hardware queue.

Discontinuities in the audio output, off-pitch notes, and off-beat notes, are all very annoying even to the uneducated human ear and are therefore not tolerable. Thus, *hard* ATLAS requirements include that all task deadlines which affect voice pitch, voice synchrony, and the steady flow of digital samples must always be met. The *soft* requirements are that the quality of the produced music be *best-effort*, i.e. that for every system state, the system must always select the synthesis modes that maximize the overall SNR (signal-to-noise ratio) without jeopardizing the completion of any hard deadlines. For any given platform, one can always devise a system state, i.e. a large enough number of concurrent high-pitched voices with very demanding synthesis methods, which will overload that platform even when using synthesis methods with the lowest bandwidths. Lower volume voices contribute less to the SNR, so lower volume voices will be shed until the hard deadlines of all remaining voices can be guaranteed.

Each concurrent voice is treated here as a *job*, the synthesis method options available to each voice are treated as job *modes*, and the number and nature of concurrent voice jobs is clearly dynamic. As indicated by Figure 1.2, besides the dynamic number of voice jobs there are several additional jobs in the system responsible for MIDI processing, interactive graphic display, and the time-stamped recording of the internal trace events necessary for verifying and evaluating system performance. The top-layer architecture must facilitate the online best-effort selection of voice job modes, the dynamic load-balancing of the varying number of jobs over the available processors, and the real-time time-sharing of the job subsets sharing each of the processors, and it must do so reliably on a diverse range of platforms.

The ATLAS application actually poses several additional challenges to our top-layer approach. Somewhat contrary to our model assumptions, ATLAS state transitions are actually quite frequent, forcing us to carry out meta-controller decisions several times a second. Real-time scheduling must therefore be resolved in a manner which can accommodate synthesis mode transitions at such high rates. Furthermore, voice synthesis mode transitions cannot be carried out while a note is being generated, so that each meta-control decision must consider a different set of voice mode alternatives for each decision epoch. Finally, when dealing with *hard* real-time applications, our top-layer approach generally assumes WCETs for all tasks and it assumes that all platforms are such that task WCETs are readily derived for each platform. We believe, however, that our top-layer approach has great potential for accommodating non-deterministic platforms even when assuming sub-WCETs, thereby significantly increasing system performance for such platforms. Currently targeted ATLAS platforms include non-deterministic PC and Silicon Graphics platforms so that this potential might be explored and verified. The successful application of our top-layer approach to this application would therefore serve also as an indicator regarding the flexibility of the approach and the hardness of assumed limitations.

A preliminary, non-preemptive, uniprocessor version of ATLAS was first developed and tested on the author's 33 MHz 486 and then binary-ported and run on the advisor's Pentium machine as well. A standard MIDI file served to emulate online MIDI input to ensure that identical experiments could be carried out on both machines. Continuous output sample streams and precise voice pitches were maintained on both machines, but the SNR was much higher on the more powerful PC platform. When run on weaker PC platform, only the higher volume voices were generated. The online ear and the offline evaluation of the recorded trace files confirmed that no hard deadlines were missed and that the meta-controller decisions were indeed 95% optimal for each of the machines. Thus, the top-down developed program ported very nicely to several platforms within a predetermined family. Furthermore, the current ATLAS application has been implemented in a manner which should enable it to recompile, as is, and run equally well on any Silicon Graphics platform, thereby illustrating that top-layer applications can readily port to additional platform families, as well.

1.5 Accommodating Non-Determinism

As already mentioned, in the course of ATLAS development, we had to overcome non-deterministic task behavior due to platform elements such as memory and disk caching, as well as underlying device driver processes for the mouse, MIDI, and disk recording. The deterministic accommodation of non-deterministic platform and program elements is a major concern in the hard real-time context, primarily today when there is such a strong demand to abandon the development of costly deterministic real-time platforms and make do with COTS (Commercial Off-The-Shelf) platforms, which are primarily non-deterministic. It is well known [MW94] that accurate WCET (worst-case executions times) are elusive for such platforms, while pessimistic WCET assumptions result in platform utilizations that are prohibitively low. This spurred the development of several novel techniques that enable us to guarantee all hard deadlines without *always* assuming pessimistic WCETs. These techniques were incorporated into ATLAS and they appear to have been successful on our PC platforms. Our research in this area has only begun, and we have yet to rigorously experiment and analyze our results, but our initial experience and several intuitive insights have led us to believe that the top-layer approach does indeed have much to offer in this respect.

1.6 Thesis Structure

Following is the structure and contents of the rest of the thesis, motivating each chapter by elements of the discussion above.

- Chapter 2 focuses on previous work. It begins with a brief survey of the classical real-time scheduling policies which provide the basis for, and are readily incorporated into, the top-layer architecture and then follows with a survey of comparative conventional approaches which deal with various sub-problems within our problem domain. A survey of commercial and experimental real-time platforms is also provided to point out existing elements which might serve our approach, and to better understand the practical issues which must be overcome.

- As previously described, the key to our top-layer approach are simple, yet flexible, platform and program models that increase the software portability and greatly reduce the complexity of software meta-control. These models and their associated control algorithms are then incorporated into a top-layer software architecture to obtain fully portable real-time software. This architecture is formally presented in Chapter 3, along with a several examples that demonstrate how these models can be applied to realistically complex real-time systems.

- Having described the software architecture, we then venture into the control algorithms needed to facilitate this architecture. In Chapter 4 we discuss job time-sharing algorithms, also formally establishing the theoretical and practical foundations of EDF job time-sharing. Here, we show how job bandwidths are computed, how a job time-sharing mechanism might be implemented, how temporal considerations within the jobs are passed on to this mechanism, and how this can serve concurrent real-time applications as well.

- Having resolved job time-sharing for each shared processor, Chapter 5 deals with job load-balancing issues, i.e. the partitioning of jobs over processors in a manner which guarantees their time-shared schedulability. We begin by showing that job load-balancing is essentially a generalization of the classical binpacking problem and therefore readily resolved using conventional binpacking approximation policies. We then address the issue of estimating the probability that a given load-balancing policy will indeed succeed when provided with a given set of job bandwidths and processor speeds. This probability plays an important role in deciding which load-balancing policy to use and which job mode combinations ought to be evaluated when the meta-controller policies of Chapter 6 must make time-bound decisions. Here, we propose a model for estimating these probabilities for any given problem.

Another important issue is how to carry out *best-effort job-shedding* and load-balancing in an overload situation when any of the jobs can be suspended[2], i.e., how to partition a subset of jobs over processors in a manner that maximizes the aggregate rewards contributed by the jobs to the system gain. Here, we devise several alternatives for integrating conventional binpacking and density-greedy algorithms. We also present comparative results from a few Monte-Carlo experiments for an evaluation of their time complexities and performance.

- Chapter 6 addresses the issue of making a single meta-control decision at a given decision epoch. Here, we show that simultaneous best-effort job mode-selection and load-balancing is essentially equivalent to a composite binpacking and multiple-choice knapsacking problem, which is NP hard. We incorporate the load-balancing policies of Chapter 5 into two families of practical solutions devised to make cost-effective meta-control decisions. The QDP family of solutions uses bandwidth quantization, dynamic programming, and an optional Markov decision process model to make decisions that are later shown to be statistically ϵ-approximate. The gradient-greedy G^2 family achieves similar, but less predictable results, with shorter response times. Both families can be used to make time-bounded best-effort decisions. Extensive Monte-Carlo experimentation has found that both produce exceptional results with reasonable response times.

- Faced with several families of algorithms for making these meta-control decisions, a practical system must develop a *policy-making* strategy for choosing the most effective meta-control algorithm for each decision epoch. Chapter 7 develops a model for measuring the empirical performance of any given top-layer meta-control policy and shows how QDP performance can be estimated analytically for a given program when the distribution of runtime states is *a priori* known. These results are used to demonstrate that QDP is inherently statistically ϵ-approximate. We then develop the notion of *optimal* policy-making and we use a Monte Carlo experiment to demonstrate how an adaptive *follow-the-leader* scheme can produce optimal performance when the distribution of runtime state is *not* known a priori.

[2] This *best-effort job shedding* problem is essentially a special case of the dynamic program model when each job has only a single non-suspend mode along with a suspend mode, whereupon the system is required to run those jobs that maximize the system gain and suspend the rest.

- To demonstrate the feasibility of our approach, Chapter 8 provides a detailed description of the ATLAS case study, including the system requirements, the long-term design, and the initial implementation. It is here that we show how various limitations of our model can be overcome in a realistic application, and how anticipated rewards are actually reaped.
- Chapter 9 is devoted to the very burning issue of how to accommodate non-deterministic platform behavior in the hard real-time context without always assuming pessimistic worst-case execution times for all tasks. As previously described, we believe that the top-layer approach has much to offer in this realm and many of our ideas have been successfully incorporated into ATLAS for evaluation, but a comprehensive evaluation has yet to be carried out. Our ideas and initial experience is brought out here in the hope that they might serve future research in this area.
- Chapter 10 evaluates the strengths and limitations of our model and compares it with other comparable approaches.

The details provided by each of these chapters should serve as a springboard for more extensive efforts to provide comprehensive top-layer solutions for the anticipated needs of next-generation hard real-time systems.

Chapter 2

Previous Work

This chapter focuses on previous work. Stankovic and Ramamritham [SR90] were the first to suggest that design complexity might be reduced and worst-case assumptions might be relaxed by using a *top-layer* design approach, i.e., an approach which attempts to achieve predictable real-time behavior while focusing primarily on the program layer. Furthermore, they suggest that when dealing with many *next generation* real-time systems, a top-layer approach might be the only viable alternative. To date, no one has demonstrated that such an approach can indeed accommodate design complexities such as those targeted by this thesis.

The purpose of this chapter is to survey existing policies and techniques which serve our top-layer approach, and to describe comparative alternatives which will serve as yardsticks for evaluating our approach. Thus, this chapter helps place our work within its proper historical and scientific perspective. We begin in Section 2.1 with a brief survey of the classical real-time scheduling policies which provide the basis for, and are readily incorporated into, the top-layer architecture. Section 2.2 then follows with a survey of comparative conventional approaches which deal with various sub-problems within our problem domain.

2.1 Real-Time Scheduling Policies

This section provides a brief introduction to the classical real-time scheduling policies which provide the basis for, and are readily incorporated into, the top-layer architecture. The term "policy" is used here rather than "algorithm" because a scheduling policy will often entail using a set of algorithms to be used at different phases of application design and execution. For more extensive surveys on real-time scheduling, we refer to [SC88, HS91, XP93, SSNB95].

2.1.1 Real-Time Scheduling Objectives

As already described in Section 1.1, real-time applications in this thesis are viewed as a set of perpetual tasks. Given a set τ of tasks, a schedule is *feasible* if all task instances in τ can be scheduled such that all timing constraints and resource requirements are met. For a given scheduling policy, δ, a set τ is *schedulable* by δ if that policy can produce a feasible schedule. Scheduling policies, then, must generally address the following three objectives: schedulability analysis, schedule construction, and run-time dispatching. *Schedulability analysis*, also referred to as *schedulability* or *feasibility testing*, determines whether a feasible schedule exists for a set of tasks, whereas *schedule construction* calculates a feasible schedule if it exists. *Run-time dispatching* then transfers processing control to appropriate tasks at their schedule-appointed times. Whenever possible, schedulability analysis and schedule construction is carried out *offline*, i.e. prior to run-time, otherwise it is generally referred to as *online* scheduling. As we will see, schedulability analysis and timely run-time dispatching can often be carried out without schedule construction.

2.1.2 Static Versus Dynamic Scheduling

Most classical scheduling theory deals with *static* schedulers which have complete knowledge of task sets, characteristics and constraints, including all future release times [SSNB95]. This set of assumptions is realistic for many real-time systems, e.g. a simple process-control application having a fixed set of sensors and actuators, operating in a fixed environment with fixed requirements. For such applications, a static scheduler can then produce a fixed periodic schedule which will satisfy all system requirements having considered all task characteristics and constraints. Even when task sets and characteristics vary, a static scheduler can be used by considering the worst-case set of tasks and assuming worst-case characteristics.

When lacking complete *a priori* knowledge we must apply a *dynamic* scheduling policy to reschedule tasks as task sets, characteristics, and constraints become known. While *all* static schedules will typically be computed *offline*, dynamic schedulers will typically execute *online*. Nevertheless, we will soon see that there are cases where schedulability analysis and scheduling policy decisions can be carried out offline even when dynamic scheduling is required, so that online dispatching overhead can be minimized.

2.1.3 Optimal Real-Time Scheduling

As already described in Section 1.1, the notion of *optimal* real-time scheduling varies with the context. In a *static* scheduling context, a policy, δ, is said to be *optimal* for a given application, τ, if no other static policy can schedule τ when δ cannot. For a given *dynamic* scheduler, a newly arriving task is said to be *schedulable* if it can be scheduled to meet its timing constraints without violating the schedulability of previously scheduled tasks. Consequently, an dynamic scheduling policy, δ, is said to be *optimal* for τ if no other policy can schedule an arriving task when δ cannot [SZ92]. When tasks are optional and not all arriving tasks are schedulable, a dynamic policy might be considered optimal when it maximizes the guarantee ratio of arriving tasks, or alternately, when it schedules those tasks which maximize the system *gain*, as defined in Section 1.1.

Task sets which are periodic, preemptive, independent (non-blocking), with known fixed characteristics, and without precedence relationships, are henceforth referred to as task sets with *simple* task characteristics. Let $\hat{\tau}$ be an arbitrary task set in which all tasks in $\hat{\tau}$ are hard and with simple task characteristics. Following are three *classic* dynamic scheduling policies that have been shown to be optimal for $\hat{\tau}$ task sets on a uniprocessor.

2.1.4 Rate-Monotonic (RM)

Liu and Layland [LSD89, LL73] have shown that dynamic *rate-monotonic* (shortest-period-first) scheduling is optimal amongst fixed-priority schemes. It is quite popular because most platforms have efficient, low-level, priority-based, task dispatchers, and no additional scheduling overhead is required. It also offers a simple, conservative processor utilization-based formula for guaranteeing scheduling feasibility. It has been established, for example, that when all n tasks in task set, τ, are independent, then τ is always RM schedulable if the processor utilization, $u(\tau)$, (see Equation (1.1)), does not exceed the following function of n:

$$u(\tau) \leq n(2^{1/n} - 1). \tag{2.1}$$

As the number of tasks goes to infinity, the utilization upper bound of Equation (2.1) approaches $\ln(2) = 0.693$. Thus RM scheduling is also *utilization-*

bound, i.e. limiting the schedulable processor load using this method. But as previously noted, this utilization upper bound formula is *conservative* in the sense that task sets with higher utilizations might still be RM schedulable. In *harmonic* task sets, every task period is a multiple of all higher-frequency tasks, whereupon τ is RM schedulable even when $u(\tau) = 1$.

2.1.5 Earliest Deadline First (EDF)

Dertouzos [Der74] showed that *earliest deadline first* scheduling is optimal for the dynamic scheduling of $\hat{\tau}$, but only when a feasible schedule exists, and when run-time overheads need not be addressed. When dealing with periodic and independent tasks, full processor utilization can be achieved, i.e., all τ task sets are EDF schedulable if $u(\tau) \leq 1$. Its performance is poor for unschedulable tasks because it will attempt to meet unattainable earlier deadlines at the expense of later higher-value deadlines which can be met [Loc86].

2.1.6 Smallest Laxity First (SLF)

Others [Mok84, DM89] have shown that *smallest-laxity-first* scheduling is another optimal online policy for $\hat{\tau}$, in which laxity, or slack, is the excess time, $D_i - C_i$. This scheme is more demanding, and usually results in more task preemptions than necessary, but negative slacks provide us with an *early warning* that deadline faults are impending. With this scheme, higher guarantee ratios (see 2.1.3) are obtainable when $\hat{\tau}$ is not schedulable.

2.1.7 Cyclic Executive (CE)

Probably the most common *static* scheduler is the *cyclic executive* (CE) [BS88a, Loc92], whereby each task is allocated specific preallocated time frames, often referred to as CE *partitions*, in a cyclic schedule. CE is non-preemptive, so the tasks must further be broken down into atomic subtasks with known WCETs which fit into the allocated time partitions. For the above $\hat{\tau}$ task set, one straight-forward way to construct such a schedule would be to choose a CE period equal to the LCM (least common multiple) of the $\hat{\tau}$ task periods, and then offline approximate RM, EDF, or SLF scheduling to generate subtask time frames within the larger LCM time period. A CE dispatcher can then use a cyclic list of subtask identifiers to activate each subtask in turn with possible calculated delays between subtasks to keep them on schedule. CE dispatching overhead is therefore minimal since no task preemption is required.

2.2 Complex Real-Time Solutions

As described in Section 1.1, the targeted problem domain of this thesis are real-time applications with dynamic, state-dependent, task sets and task characteristics, capable of running concurrently with additional real-time applications of a similar nature on a variety of shared-memory multiprocessor platforms. We assume that the application states are *a priori* known and vary only occasionally. When provided with multiple modes of operation, we expect these applications to online select these modes to achieve some notion of best-effort performance. For simplicity, we assume that non-computational resources are abundant enough to be disregarded when load-balancing, though the models developed here are readily extended to support multi-dimensional resource limits, as well.

We further assume that the state-dependent task sets may have *complex task characteristics*, which throughout this thesis will imply both hard and soft deadlines, periodic and sporadic tasks, non-preemptible code segments, and task inter-dependencies which can cause one task to block the execution of another task. When dealing with complex task characteristics, the above CE, EDF, RM, and SLF scheduling policies cannot be so simply applied. Nevertheless we do assume that for each state there exists an *a priori* known scheduling policy and computational capacity such that the scheduling policy can guarantee all hard deadlines while maintaining an acceptable system gain for all soft deadlines when running alone on a uniprocessor with that capacity.

For the sake of brevity, throughout this thesis, such applications are referred to simply as *complex* real-time systems. Two brief examples of such applications appeared in Section 1.3, several additional examples are later provided by Section 3.3, and the detailed ATLAS example is the subject of Chapter 8.

We know of no conventional approaches which can simultaneously cater to so many degrees of complexity, but there are several approaches which cater to various *subsets* of these complexities. These approaches are the subject of this section.

But before we proceed, a few clarifications are in order. Firstly, all *hard* tasks in our applications, i.e. all tasks with hard deadlines, are assumed to be *mandatory* unless otherwise specified. Applications with hard tasks should therefore be refused if *any* of the hard deadlines cannot be guaranteed. Secondly, each task has a known WCET for each application state which also includes exception handling. Thus schedulers and meta-controllers may assume that these WCETs will never be violated as long as the application remains within the current state.

2.2.1 Complex Uniprocessor Scheduling

We now survey several practical approaches to dynamic task sets with *complex* task characteristics when running on a single processor. As already described in Section 2.1.3, classical RM, EDF, and SLF task scheduling algorithms have been found to be optimal for dynamic uniprocessor real-time systems when all tasks have *simple* task characteristics, i.e. periodic, preemptive, independent and with a fixed computation time. Unfortunately, such simple task characteristics are generally rare. Variants and additional optimal polynomial-time algorithms have been developed for a variety of more complex characteristics with appropriate assumptions [SC88], but these exceptions demonstrate that for a growing range of complex real-time applications optimal scheduling algorithms are either intractable or nonexistent, even when run on a *single* processor. Mok [Mok84] has shown, for example, that dynamic scheduling cannot be optimal when there are exclusion relationships. Even when assumptions are made such that optimal algorithms exist, they usually cannot be applied online to dynamic task sets due to intractable computation complexities or intolerable run-time overheads.

Consequently, as real-time systems grow more complex, research in this area is shifting towards finding practical *suboptimal* scheduling solutions for such systems. According to Stankovic [Sta93], such solutions are readily classified as belonging to one of the following three approaches: offline planning, online planning, and on-demand. To these we also add a hybrid scheduling approach, highlighting the advantages and disadvantages of each approach, and showing that, within our complex real-time problem domain, all of these practical approaches can be useful for resolving uniprocessor real-time scheduling problems for any given application state.

Offline Planning

The off-line planning approach assumes that all application states are foreseeable, whereby blocking is avoided by using an appropriate CE schedule for each state. This approach has been adopted by the *MARS* project [DRSK89], because it is highly deterministic, it has low run-time overhead and its run-time behavior is readily analyzed and understood. *Optimal* static schedulers are known to be NP-hard [BS88a], but being off-line, this may still be feasible for small enough applications. Various effective heuristics, such as RM and EDF, have been devised to reduce the *offline* scheduling complexity to polynomial levels. It has been found [XP93], for example, that EDF strategies generally reduce run-time overhead by requiring fewer preemptions.

Still, the off-line planning approach can usually be applied only to applications with a reasonably small number of tasks and states. Furthermore, its rigid, periodic nature has the potential for extremely low resource utilization and high cost. Tasks, for example, are not always readily split into partition-fitting subtasks. Time-driven sporadic tasks must be treated as periodic tasks with preallocated periodic partitions which are largely wasted. Reaction time to interrupts can be low, because event-driven servicing must be isolated within separate subsystems which queue requests for eventual handling by periodic handlers within predetermined frames. State transitions can be cumbersome because schedule switching may require partition-dependent cleanups and initializations. Performance is drastically reduced in unforeseen environmental circumstances and in unexpected task overloads.

Online Planning

The online planning approach attempts to produce block-free schedules for tasks during run-time and is therefore able to accommodate a much broader range of complex real-time systems with varying task characteristics and unpredictable application states. Each platform resource maintains a *calendar* of time frames reserved for already accepted tasks, whereby incoming tasks or task instances are accepted only if all of their resource requirements can also be satisfied without jeopardizing guarantees for already accepted tasks. This approach has been adopted by the *Spring* [SR89] and *MARUTI* [LTCA89] projects because it is highly flexible and generally more efficient.

Still, online planning shares many of the offline planning difficulties regarding sporadic tasks and reaction times. Most acute of all problems is run-time overhead, which can be very significant even when using heuristics algorithms such as *limited backtracking* [ZRS87]. In an industrial processing plant, for example, each process has a predetermined duration, whereupon this approach can determine which of the awaiting processes can currently be carried out without disrupting processes already begun. In our problem domain, however, we assume that tasks are perpetual, since we have no idea when the application state will change. When dealing with hard, mandatory, and perpetual task sets, we would require the calendar to cover planning periods equal to the LCM of all task periods, and this is intractable for arbitrary sets of task periods and dynamic task sets. Such an approach is therefore more readily applied to soft task sets, to hard tasks which are optional (whereupon each instance is independently tested for schedulability), or to mandatory task sets which are not perpetual.

The On-Demand Approach

The online on-demand approach assigns resources only when actually requested by each task instance. The RM+PCP model is typical of on-demand approaches, in which tasks are assigned priorities *without* trying to avoid blocking, while PCP, the *priority ceiling protocol* [CL90, SRL90], prevents *priority inversion* and ensures that blocking penalties will be predictable so that it can be applied to mandatory tasks as well. This approach has been adopted by the *ARTS* [ITM92] and *Concord* [LLS+91] projects because run-time overhead is minimal, resources are handled most efficiently, sensitivity to task overloads is greatly reduced [SZ92], and conservative schedulability analysis is possible even for perpetual, hard, and mandatory tasks.

Still, task-blocking overheads can be high, resulting in low processor utilization, and PCP does not support precedence. Actual execution histories are hard to understand and reproduce, making them difficult to test and debug. And schedulability analysis overheads can be very significant when dealing with a large number of tasks, and this can be prohibitive when carried out online to address dynamic task sets. In our problem domain, however, all task sets are fixed for each application state, so that on-demand schedulability for each state can be verified offline for any given platform.

Hybrid Scheduling

What characterizes all of the above approaches is that a uniform scheduling policy is applied to an integrated set of tasks. Young and Shu [YS91] were probably the first to attempt *hybrid* static and dynamic uniprocessor scheduling, also referred to as *online/offline* scheduling. The primary objective of such a hybrid scheme is to partition a difficult scheduling problem such that the most effective scheduling solution can be independently applied to each partition. Thus a *static* CE scheduler will usually be applied to a set of strictly periodic tasks, particularly when worst-case release phasings, resource contentions, untimely preemptions, and unnecessary context switching overheads can be reduced or avoided. *Dynamic* schedulers will usually be applied to task sets which cannot be efficiently accommodated by static scheduling, e.g., sporadic tasks with irregular arrival times, and tasks which are not readily broken up into subtasks which fit into cyclic time partitions. Young and Shu dealt specifically with uniprocessing and with reconciling CE scheduling with RM scheduling. Schedule interleaving was therefore accomplished by treating each CE partition as a single task for the purposes of online RM scheduling and schedulability analysis. Thus this hybrid approach can be very useful in resolving uniprocessor real-time scheduling for the complex task characteristics of our problem domain.

2.2.2 Multiprocessor Load-Balancing

Complex real-time systems often require centralized multiprocessor clusters and distributed networks to carry out their mission. The current scope of our research is confined to the easier case of shared-memory multiprocessor platforms, whereupon application tasks must either be scheduled *globally* by a single scheduler which dispatches them to the available processors, or they can be *partitioned* over the available processors and then scheduled by a *local* scheduler on each of the processors. Distributed platforms are confined to the latter approach.

When using a global scheduler, Mok and Dertouzos [MD78] have shown that there is no optimal scheduling algorithm for multiprocessor platforms, even for independent preemptible tasks, if the arrival times of the tasks are not known *a priori*. When dealing with complex task characteristics, even sub-optimal global scheduling is rarely viable. A more common approach, therefore, is to seek an offline static partitioning of tasks, whereupon each processor can locally schedule its task subset using any of the practical approaches of Section 2.2.1. Optimal partitioning, however, is known to be NP-hard [DL78], so suboptimal partitioning solutions have been devised, include *bin-packing* partitioning for EDF scheduling and *next-fit* RM schemes. Such an approach cannot be applied to dynamic task sets, but in our problem domain, each state has a fixed task set so that this can be carried out for each application state if the number of states is reasonably small.

In the more general case, however, or when the number of application states is exponential, dynamic tasks sets require *online* allocation of tasks to processors, referred to here as task *load balancing*. Sometimes even task *migration* while executing is considered [MJ70]. Being even more complex than static partitioning, dynamic load-balancing on a multiprocessor platform is generally NP-hard. Jo [Oh94] has shown that, when treating only pure periodic task sets, dynamic load-balancing can be practically resolved via *centralized* binpacking heuristics. This, however, is restricted to very simple task characteristics.

To address more complex task characteristics, Ramamritham *et al* [RSS90] advocate a *distributed* FCFS (first-come-first-serve) approach, where an appropriate schedulability test is used used on each processor to determine whether the requirements of a newly arrived task can be guaranteed by the local dynamic scheduler without endangering the deadlines of tasks already resident on that processor. Schedulability overheads can be prohibitive for frequent arrivals, so McElhone [McE94] shows how the schedulability complexities can be traded off with pessimism. Nevertheless, online load-balancing and schedulability overheads remain very high and are usually prohibitive for large-scale applications.

In our top-layer approach, we will want to support online load-balancing so that we can cater to very large numbers of application states, but we would like to do so with significantly lower overhead. For more detailed reviews on real-time scheduling in multiprocessor and distributed systems, we refer to [ATY87, Hsu90, DL78].

2.2.3 Best-Effort Adaptation

Though not necessarily referred to as such, the notion of best adapting to varying runtime circumstances already exists in many conventional systems, primarily as a mechanism for avoiding anticipated overloads [SZ92, KM85, Car84] or as a way to deal with uncertainty in the execution environment [GS90]. There are two essential partners to best-effort adaptation. First it requires that the application be flexibly designed so that it can adapt. Secondly there must be a meta-control mechanism for deciding which adaptation is the most appropriate for any given runtime state.

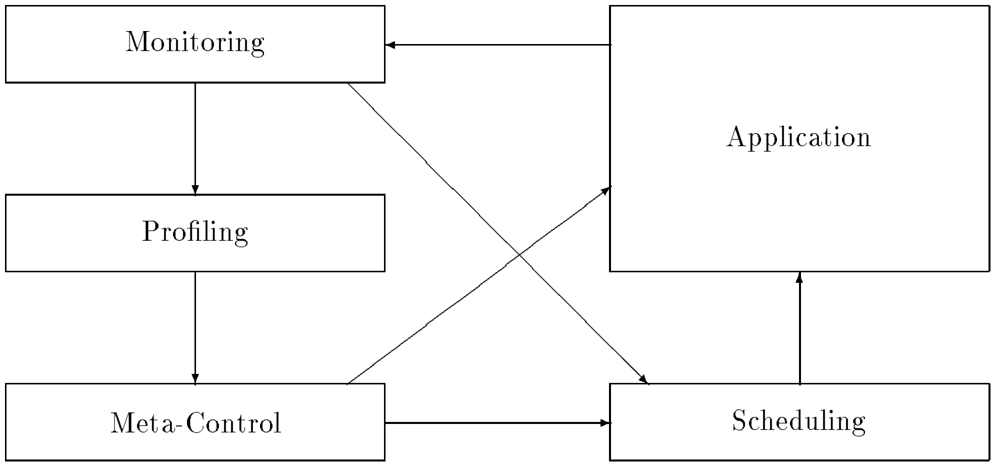

Figure 2.1: Typical Adaptive Meta-Control

Adaptation Techniques

Common forms of adaptation include *approximation* techniques which adjust time constraints, e.g. sensor and reactor rates, to supportable levels [GS89]; *imprecision* techniques which use sieve functions to produce intermediate results when time runs outs [LLS+91]; and *polymorphic* techniques which provide alternate methods for accomplishing specific tasks [KL91] or which use different scheduling policies such as altering scheduling parameters [RS91]. Additional forms which might also be considered adaptive include task migration [Bok79], dynamic redundancy management for fault tolerance [BS88b], and *variable characteristics* which provide task computation times and resource requirements as a function of internal state variables [KL91] or as function of elapsed execution time [HS89] so that worst-case assumptions need not always be made. As we can see, adaptation techniques essentially seek methods to either promote inherent scheduling flexibility, optimize scheduling efficiency, or facilitate graceful degradation when other alternatives have been exhausted.

Adaptivity Control

Preventive adaptations must anticipate problems within the system, while *reactive* adaptations must respond to internal failures or external events. Both classes of adaptations require an appropriate *meta-control* model capable of handling all required adaptations.

Figure 2.1 illustrates a typical adaptive control model in which monitored application performance is profiled, evaluated, and fed into a meta-control subsystem which may alter the application configuration and affect real-time scheduling. Observed performance may also serve ongoing scheduling activities. Time constraints, task characteristics, and meta-control directives are usually provided the meta-controller, in a variety of forms, by the application. Once provided, *milestoning* [TM89], global and local slacks [Loc86], utilization thresholds [LSD89], and other state variables can be monitored to trigger appropriate adaptations.

Ultimate Meta-Control Objectives

Current meta-control designs appear to be restricted in the following ways. First, none of the surveyed solutions offer objective criteria for measuring the performance of their schemes. We would like to know precisely how close a given solution is to some notion of *optimal*. Secondly, considering all possible adaptations is typically computationally intractable, so the scope of considered adaptations is usually limited. In the Spring kernel [RS91] for example, Meta Level Controller adaptations are confined to altering scheduling algorithms and parameters. RESAS adaptations [BS88b] include processor assignment (load-balancing) and the management of various time/space redundancies, but only one adaptation form (one degree of freedom) is actually treated at a time.

A major top-layer objective is to devise a meta-control scheme capable of automatically considering all adaptation alternatives, and to do so in a manner which enables us to objectively measure its performance.

2.2.4 Real-Time Platforms

Our notion of a real-time *platform* relates to all system components which depend upon the specific computational engine used. This notion of a real-time platform embraces all system layers which serve the application software, including languages, compilers, libraries, linkers, operating systems, processing hardware, communication channels, and all other platform-dependent tools used for development and maintenance. Platform layers do not include system elements which have been dictated by the environment, such as specialized sensors and actuators. These are viewed as part of the application *environment* so that timing constraints imposed by these elements can be related to as part of the system requirements which are platform-independent and therefore not imposed by a given solution.

Real-time platforms must make provisions for global time, for expressing deadlines and other timing constraints, for context-driven exception handling, for predictable resource management and task synchronization, and for real-time scheduling. Adaptive real-time platforms must accommodate all aspects of adaptive meta-control as well.

These needs have spurred the development of a large selection of real-time languages, operating systems and architectures. Real-time programming *languages* include Real-time Euclid[Sto87], PEARL[HS90], RTC++[ITM92], Real-Time Concurrent C[Ram91] and FLEX[KL91]. All industrial real-time *operating systems* such as RMX[1], VRTX[2] [Rea86], Real-Time UNIX [FGG+91], LynxOS[3], and VxWorks[4], provide a wide range of POSIX-like real-time services, but real-time scheduling is left to the user. Real-Time MACH [TNR90] is a very portable, DARPA-backed, operating system which also addresses online real-time scheduling. *Adaptive* operating systems include the Spring kernel [SR89], the ARTS kernel [TM89], RESAS [BS91], GEM [SBWT87], CHAOS [GS89, SGZ90, GS90], Dragon Slayer [WAB+89], and MARUTI [LTCA89]. At the hardware level, we require customized processor nodes and buses, clock synchronization units, and deterministic networks such as LANCE [DRSK89], SpringNet [SR89], and the IEEE 802.5 Token Ring [ITM92]. Typical real-time platforms are designed to be readily expanded to accommodate more nodes and resources if necessary.

All of the above platforms illustrate how differences in the approach touch upon every aspect of the platform architecture. The Spring project, for example, uses an *online planning* approach to real-time scheduling. Its customized real-time platform includes *Real-Time Concurrent C* as the programming language, the *Spring Kernel* as the real-time operating system, and the *Spring Net* as the real-time network. To reduce the runtime overheads of *online planning*, customized co-processors have been developed to facilitate the real-time scheduling process. All tasks must be accompanied by

[1] Intel Corporation
[2] Hunter & Ready, Inc.
[3] Onyx Technologies Ltd.
[4] Wind River Systems

a priori known task descriptors with all task characteristics including worst-case execution times, deadlines, criticality, preemptivity, required resources, precedence graph, synchronous/asynchronous communication graph, location of task copies, real-time or not, and incremental or not. Varying characteristics are not supported, so worst-case assumptions are always made. Application tasks are made core-resident on all processor nodes before they can be invoked with hard deadlines. Event-driven processing is isolated in separate subsystems. Local and global schedulers cooperate in a dynamic guarantee algorithm [RS84] which avoids resource contention. Tasks are either mandatory or optional with incremental task portions being treated as optional. A *Meta-Level Controller* anticipates real-time exceptions and carries out preventive adaptations by modifying scheduling parameters.

Thus, all platform layers are highly customized and non-interchangeable with other platforms. Each platform normally has its own approach which functionally affects all layers, including those of the application software. With such diversity, and with no accepted standards, it is not surprising that, as far as we know, no adaptive real-time platforms are commercially available today.

Another major limitation with these architectures is that they require *white-box* applications, i.e. applications capable of providing complete knowledge of all internal task characteristics so that the schedulability of tasks and various adaptation choices can be assessed. Such a requirement might not be feasible for many commercial or large-scale applications. We prefer a *black-box*, or object-oriented, architecture where nearly all internal characteristics are hidden and only a minimal set of application parameters must be divulged. And another major difficulty is that the distinction between meta-control and scheduling layers is not always apparent. Note, for example, that imprecision, polymorphism, variable characteristics, and scheduling policy optimization are all treated here as adaptive techniques even though they are sometimes directly integrated into the scheduler. This coupling greatly increases meta-control complexity and impedes upon the development of generic meta-control models.

Major top-layer objectives include support for *black-box* applications and the development of a layered control architecture with well-defined generic interfaces between the scheduling and meta-control layers. Another major objective is to devise a top-layer architecture capable of running atop any of the commercial platforms.

2.2.5 Real-Time Portability

A significant design objective for any system is the ability to accommodate *long-term* adaptations, e.g. evolving run-time platforms due to technological advance, and evolving program specifications which change with time and experience. Commercial real-time languages and operating systems, e.g. those surveyed in the previous subsection, provide a basic level of portability by supporting source-level or code-level compatibility between a significant choice of real-time platforms, at least from a *functional* point of view. From a temporal point of view, however, conventional real-time design methods are inherently non-portable, because scheduling and system analysis begins only *after* all application and underlying platform task characteristics have been fully established and considered. Porting to a new platform with different underlying platform tasks and different application task characteristics may require extensive re-engineering. A top-layer approach seeks to attain a higher level of portability called *real-time* portability, which enables an application to port from one platform to another without having to re-tailor its scheduling, load-balancing, and best-effort meta-control policies. In this context, we cite here two comparable efforts to achieve real-time portability, that of the Concord project, and the very recent birth of PERC.

The Concord Approach

The Concord project [LLS+91], provides a fully portable model for adaptive real-time applications, provided that task characteristics are relatively simple and flexibility is obtained only by imprecision. Thus each task is comprised of mandatory and optional activities, mandatory activities are scheduled by online on-demand policies, and optional activities are scheduled at a lower priority using a best-effort value-based scheduling policy. Comprehensive real-time monitoring and task profiling schemes enable us to model the computational requirements of each activity as a user-constructed function of variables which are automatically derived for each platform and refined with accumulated experience. Task characteristics are thus automatically determined for each platform, whereupon these applications can reliably execute on any platform which can guarantee the timely execution of the mandatory activities, while exploiting remaining platform resources to maximize the system gain produced by the optional activities.

This is precisely what we wish to achieve in our top-layer approach but with a greater degree of flexibility, lower online overheads, objective performance measures, and with the ability to cater to black-box applications with more complex task characteristics.

The PERC Approach

Many of the above principles are recently being incorporated into the PERC project [Nil95], currently being implemented by Kelvin Nilsen at NewMonics Inc. PERC stands for the *portable executive for reliable control* and it is essentially an extension of the Java programming language and the Java virtual machine, designed to support the special needs of reliable real-time applications. As in Java, all PERC applications comprise of window-based subsystems called *applets*. Each applet consists of any number of mandatory periodic and sporadic threads, as well as any number of optional event-driven threads called *spontaneous* threads and non-real-time threads called *ongoing* threads. As in Concord, all mandatory threads have bound execution times coded within them, and they consist of mandatory activities followed by any number of optional activities to be carried out if time permits. PERC supports concurrent real-time applications, so the set of concurrent applets and subsequent threads is not *a priori* known. The PERC runtime executive must therefore provide a reliable real-time scheduling policy which will always guarantee the timely execution of accepted mandatory threads while also providing reasonable criteria for deciding which incoming applications must be rejected due to lack of resources. PERC offers two possible scheduling solutions to this problem. When the LCM of all thread periods is reasonable, PERC proposes using a static CE schedule which is constructed online using an RM-based heuristic. Otherwise, PERC uses a dynamic RM-based scheduling policy.

PERC offers no global best-effort meta-control mechanism, but it does provide each application with the necessary profiling services for evaluating optimistic, typical and pessimistic execution times for the thread activities. Each applet is expected to include an *administrator* thread which will use these services to compute minimal and desirable execution times for each thread, will then *negotiate* resource budgets with the real-time executive, and finally internally adapt to best exploit the resources allocated to it. The budget negotiating protocol is uniform for all applets, letting each applet first state the required thread frequencies and their minimal and desirable execution times, whereupon the real-time executive must provide whatever it can, possibly even asking already executing applets if they can release some of the resources already allocated to them. It is the implementor's responsibility to provide "responsive" performance to arbitrary resource budgets. Resource budgets are continually reevaluated with variations in the set of applets and in their internal states. Budget negotiating activities are carried out by dialogues between the real-time executive and the activity administrators which can be initiated at any time by any of these parties. This approach introduces a FCFS bias, since earlier applications will usually get more resources than the later applications, but cooperating applications will enable new applications to join them whenever possible.

As far as real-time portability is concerned, PERC portability is ultimate in the sense that PERC applications are portable at the binary level, capable of running on any PERC virtual machine which has adequate resources to accommodate the minimal mandatory thread requirements. PERC also shows how exception handling should be accounted for within the task WCET, and how interrupt-driven activities can be supported in a platform-independent manner. In comparison with the Concord project, the PERC meta-control architecture is more flexible, letting each application adapt in arbitrary ways to a given budget. This PERC negotiating process, however, does not support a global notion of best-effort performance which would force a lower gain application to yield resources to higher gain applications. Furthermore, negotiation dialogues can be lengthy and administrator response times are not *a priori* known. This might be sufficient for applications in which internal state transitions are initiated by a patient user, but it can be very inadequate for applications in which the response time to state transitions is bound. Moreover, we still lack support for objective performance measures, black-box applications, and complex task characteristics.

PERC is only now being implemented, and it has yet to be proved adequate for *hard* real-time due to the complexities of real-time garbage collection, cache management, and evaluating WCETs by empirical profiling. We nevertheless believe that there is much to be learned from this approach which comes quite close to achieving several of the goals of this thesis.

2.3 Summary

A significant variety of conventional approaches have already been devised for various subsets of our targeted complexities. These include scheduling solutions for complex task characteristics on uniprocessors, and dynamic load-balancing and real-time portability for relatively simple task characteristics. They also include best-effort architectures which lack absolute performance measures, and which are typically limited in variety of adaptation techniques and in the number of automatically considered alternatives. As indicated by the diversity of algorithms and approaches, there is no *silver bullet* for complex real-time scheduling, load-balancing, best-effort adaptation, and real-time portability, particularly when running on multiprocessor platforms. The problems are generally NP-hard and each practical solution has significant limitations. The large-scale AIMS system [CDHC94], for example, serving the monumental Boeing 777, has several multiprocessor platforms serving a diverse mix of tasks with varying hardness and criticality, and state-dependent execution times. None of the conventional approaches were found to be appropriate, and a hybrid mixture of *ad hoc* methods had to be devised and applied.

We posit that the primary source of computational complexity in our problem domain is the need to seek adequate online scheduling solutions for dynamic task sets with complex task characteristics. We therefore seek methods for reducing these scheduling complexities by decomposing our applications into smaller scheduling problems which can be resolved offline and encapsulated within task aggregates which we call *jobs*. It is this job-oriented approach which enables us to greatly reduce the complexities of online scheduling, load-balancing and best-effort adaptation. It helps cater to concurrent hard real-time applications while supporting a black-box model of these applications and providing objective measures for best-effort performance.

Chapter 3

A Top-Layer Architecture

Conventional approaches to real-time scheduling seek *optimal* solutions, and therefore attempt to address the entire system task set in an integrated manner. So when running on a multiprocessor platform, the individual task also becomes the basic unit for allocation. Load balancing, then, must not only consider task placement implications to task scheduling, but it must also consider how it will effect inter-processor communication bandwidths and basic task characteristics such as worst-case execution and maximum blocking times. As systems grow more complex, i.e. as task sets and characteristics grow more diverse and dynamic, the quest for optimal solutions become more and more futile, yet programs are still generally modeled as a single integrated task set.

To greatly reduce problem complexity, our top layer approach prefers a hierarchical aggregate approach to program modeling, whereby system designers are encouraged to structure the program as a set of loosely-coupled *jobs*, each job comprising an arbitrary state-dependent task set with an independent scheduling solution. As we show in subsequent chapters, this *divide and conquer* strategy greatly simplifies online real-time scheduling, load balancing, and best-effort adaptation, paving the way to a layered software control model. Well-designed jobs can even provide near-optimal best-effort performance. Together, the hierarchical program model and layered control model qualify as a *software architecture* in the sense that they facilitate orderly construction of a program in a manner which reduces the number and complexity of the interconnections between software components. It also offers a well-structured framework for reasoning about the various aspects of the system, understanding its behavior, and analyzing design optimality.

This chapter introduces the top-layer architecture and provides several examples to illustrate how it could be applied to a variety of applications. Subsequent chapters deal with the algorithmic support for this architecture. Section 3.1 begins with a description of the kernel RtTS architecture which is then extended in Section 3.2 to produce a more portable top-layer super-architecture. RtTS already contains all the necessary ingredients for job time-sharing, load-balancing, and online best-effort adaptation, so that it can be used, as is, when the platform family has already been established and job characteristics are already known for the processor types of that family. The additional top-layer extensions are designed to support application portability even when the platform family has not been established. The flexibility of this architecture is demonstrated in Section 3.3 which provides a variety of complex real-time applications which could benefit from such an architecture.

3.1 The RtTS Architecture

The primary RtTS objective is to minimize online complexity. The primary source of complexity in our concurrent, complex problem domain is the need to resolve real-time scheduling for arbitrary mixtures of concurrent applications, an exponential rowth in the number of application state combinations, a different set of complex task characteristics for each combination, and an exponential number of possible runtime platforms. When dealing with best-effort applications, we must further consider all possible combinations of the available mode adaptations as well. Thus the *offline* resolution of real-time scheduling is intractable due to the enormous number of solutions which we would have to compute in advance and make instantly available during runtime. *Online* resolution is intractable due to the assumed complex task characteristics, the multiprocessor platforms, and the hard deadlines which must be guaranteed for all possible application states once the application is accepted.

Job aggregates, however, can be designed to contain a reasonably small number of internal state and mode combinations. The complex uniprocessor scheduling approaches of Section 2.2.1 can then be used to *offline* compute a feasible static or dynamic scheduling solution for each internal combination when running on a dedicated uniprocessor of a predetermined type at a minimal processing speed which is determined as an integral part of the solution. The scheduling solutions for each job are then encapsulated within to produce *black-box* jobs with a standardized protocol for sharing only the minimal amount of information necessary to support job time-sharing, load-balancing, and best-effort adaptation.

The following RtTS architecture then enables us to time-share and load-balance an arbitrary set of such jobs, and these jobs may belong to an arbitrary set of concurrent applications. When dealing with best-effort applications, the RtTS architecture enables us to best-effort adapt to each state transition within the applications or the platform, provided that these transitions occur at a reasonably low rate. The algorithms necessary for these activities and their complexities are the subject of subsequent chapters. This section begins with the RtTS platform and program models in Sections 3.1.1 and 3.1.2, followed by a description of the layered RtTS control model in Section 3.1.3, and concluding with the black-box paradigm in Section 3.1.4.

3.1.1 The Platform Model

As already described in Section 1.3.1, we will generally assume that all targeted platforms belong to the same *family* in the sense that task execution times on any given processor within the family are assumed to be equal to the number of machine cycles needed to complete the task, divided by a processor characteristic called the processor *capacity*. Consequently, this processor capacity essentially represents how quickly tasks can get completed. All platforms within a family are assumed to differ only in the number and capacities of the shared-memory processors in each platform, so a given platform is readily characterized as a set of processors, C, with a processor capacity, β_c, for each processor $c \in C$.

As in many families, we hereby distinguish between three levels of family circles: the *immediate* family, an *extended* family, and *arbitrary* families, i.e. arbitrary sets of platforms. When dealing with *hard* applications, we will always assume that the targeted platforms belong to an immediate or extended family. When dealing with *soft* applications, we would like to cater to arbitrary sets of platforms, as well. As we now show, each of these circles also implies a different degree of portability.

An Immediate Family

An *immediate* family of platforms is a set of platforms which share a common machine language and which guarantee that the *relative* execution time of each machine instruction, i.e. the instruction execution time divided by a known machine-dependent function of the machine cycle time, is identical for all processors in the family. Let α represent a set of processor types with a common machine language, in which each processor type, $\alpha_i \in \alpha$, has a deterministic execution time, $t_i(n, f)$, for each machine instruction, n, when running at an arbitrary machine cycle rate, f. Then all α-based platforms constitute an immediate family of platforms if for each processor type, α_i, there exists a factor, $x_i(f)$, such that $t_i(n, f) \cdot x_i(f)$ is always equal to a constant, $k(n)$, of machine cycles on all processors in the platform family, i.e.,

$$\forall f, n, i \exists x_i(f), k(n) : t_i(n, f) \cdot x_i(f) = k(n). \tag{3.1}$$

Consider, for example, a processor type α_1 with deterministic instruction execution times expressed as a constant number, $k(n)$, of machine cycles per instruction for all f in the immediate family, then $x_1(f) = f$. Let α_2 be a different processor which emulates the same machine language and is *always* 10 times slower than α_1, then $x_2(f) = 0.1 \cdot f$. Platforms with arbitrary mixtures of α_1 and α_2 processors at arbitrary machine cycle rates clearly constitute an immediate family.

The processor capacity, β_c, for any processor, c, on any of these platforms is therefore equal to $x_i(f)$, where i identifies the processor type, α_i, and f is the processor machine cycle rate. Thus processor capacities are most naturally specified in MHz units. The execution time, C_i, of any task, i, on any processor, c, can then be computed by dividing an *a priori* known task-dependent constant, n_i by β_c, i.e.,

$$C_i = \frac{n_i}{\beta_c}. \tag{3.2}$$

The computational resources of each platform are therefore fully characterized by the set of processor capacities, $\{\beta_c\}$. Even when the number of α processor types is severely limited, the variety of processor capacities can be significant, and the number of shared-memory processors on any given platform is theoretically unlimited, so that platform family sizes can be arbitrarily large. Thus defined, immediate platform families essentially support *binary*-level portability, i.e. just load and run a top-layer application on any of the family platforms and it should execute reliably and best-effort, as long as these platforms have at least a minimal, *a priori* known, quantity of available capacities in $\{\beta_c\}$. Note, also, that this narrow notion of binary-level portability will become less and less restrictive as Java-like applications become more prevalent.

Normalized Capacities

Referring back to Equation (3.1), we note that both $x_i(f)$ and $k(n)$ are readily scaled by an arbitrary factor and the equation remains true. It is sometimes convenient to normalize all β_c values such that the highest β_c value on any given platform is equal to one, whereupon the n_i values attributed to each task i in Equation (3.2) must also be normalized by the same factor so that this equation also remains true. Normalized capacities will appear several times in this thesis.

An Extended Family

As implied by Equation (3.1), processors which lack consistent relative instruction execution times cannot be incorporated into immediate platform families, thereby excluding cached processors with non-deterministic instruction times, micro-coded processors with different instruction times per machine, and processors with different operating system components and priorities which produce different platform overheads for operating-system services. To accommodate such processors in a *hard* real-time context, we propose the notion of an *extended* family in which the capacity of each processor is measured as a factor of the *worst-case* relative instruction execution time for all processors in the extended family. Given a set of processor types, α, and given that the WCET for an instruction n on $\alpha_i \in \alpha$ at machine cycle rate f is equal to $t_i(n, f)$, then one must seek factors $x_i(f)$ for all α_i, such that:

$$\forall f, n, i : t_i(n, f) \cdot x_i(f) \leq \bar{k}(n) = \max_{\alpha_j \in \alpha} \{t_j(n, f) \cdot x_j(f)\}. \qquad (3.3)$$

As in the immediate family, the processor capacity will be equal to $x_i(f)$, whereupon the assumed instruction execution time for any given instruction, n, will be equal to $\bar{k}(n)/x_i(f)$ and this might be significantly more than the actual time. Large variances in the distribution of $t_i(n, f) \cdot x_i(f)$ values for each n will therefore introduce a great degree of pessimism in the assumed task execution times, so x_i factors must be devised to minimize this distribution. There will always be processor sets in which large variances are inevitable, hiding significant capacities which cannot be exploited by hard tasks, raising questions as to the usefulness of this platform model for those sets of processors. Future extensions should, therefore, include a *secondary* capacity for each processor which would quantify these hidden capacities so that they can be allocated to soft and optional tasks which usually outweight mandatory hard tasks in complex real-time applications.

This notion of an extended family can further be extended to platforms which do not even share a common machine language. All we have to do is devise $x_i(f)$ factors such that Equation (3.3) holds for $t_i(n, f)$ values which represent WCETs for task n instead of instruction n. We are thus guaranteed that the actual WCET for any given task n on any processor c in the extended platform family will never exceed the assumed $\bar{k}(n)/\beta_c$ WCET. Extended families can thus support a notion of *source*-level portability over a variety of processors even in a hard real-time context, i.e. just re-compile, load and run a top-layer application on any of the extended family platforms and it should execute reliably and best-effort, subject to a minimal, *a priori* known, quantity of available platform capacities in $\{\beta_c\}$.

In an immediate family, the capacity of any given processor will always be the same for both hard or soft applications. In an extended family, however, the processor capacities may very well be greater in a soft real-time context, since we can assume *typical* task or instruction execution times for $t_i(n, f)$ rather than worst-case. Also the variance of $t_i(n, f) \cdot x_i(f)$ values for different n can be reduced in a soft real-time context, since the extreme constraints of Equation (3.3) may also be somewhat relaxed. As in an immediate family, given the processor type, α_i, and the processor machine cycle rate, f, we can immediately deduce the processor capacity on any given platform as equal to $x_i(f)$. Unlike an immediate family, the $x_i(f)$ values in an extended family may differ for hard or soft applications, causing us to suggest that future platform model extensions should include several capacity classes for different classes of workloads.

Arbitrary Platforms

Immediate and extended families can thus provide binary and source level portability for hard real-time applications. But when dealing with soft applications, the quest for *a priori* known WCETs is no longer necessary, whereupon top-layer applications should be able to port between arbitrary platforms. For such applications, the processor capacity of each processor might be determined by a generic or application-specific benchmark which would enable us to estimate the typical execution time of any given task, but without having to guarantee that this estimate will always exceed the worst-case.

This introduces a more relaxed notion of source-level portability, comparable to that provided by Concord and PERC (see Section 2.2.5), whereby compilation must also be followed by offline and/or online profiling of the

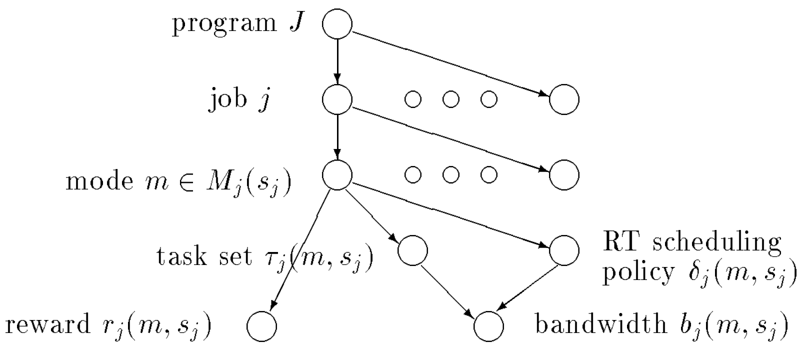

Figure 3.1: A Hierarchical Program Model

task execution times on the new platform to reevaluate the required job bandwidths. We look forward to a day when platform manufacturers will design the platforms such that precise typical and WCETs can be achieved by online profiling and a well-defined set of known platform attributes. It is this degree of portability that requires the extension of the RtTS architecture given in Section 3.2.

3.1.2 The Program Model

As previously noted, the RtTS architecture assumes the more general case where the real-time applications using it have been designed using the dynamic top-layer model of Section 1.3.3. Static applications might do with the static top-layer model of Section 1.3.2, but this is simply a special case of the dynamic model in which each job has a single state and a single mode.

As illustrated in Figure 3.1, in this hierarchical program model, the application is equal to a set of jobs, J, each job $j \in J$ having an *a priori* known set of internal job states, S_j, and each state, $s_j \in S_j$, having a valid set of alternate operation modes, $M_j(s_j)$. Each mode, $m \in M_j(s_j)$, comprises an arbitrary task set, $\tau_j(m, s_j)$, which is independently scheduled by an offline-determined static or dynamic scheduling policy, $\delta_j(m, s_j)$. We assume that this $\delta_j(m, s_j)$ scheduling policy has already been internally coded within the $\tau_j(m, s_j)$ task set[1], and that it has also been found to satisfy the hard and

[1] We recall from Section 2.2.4 that state-of-the-art real-time operating systems generally require that the real-time scheduling solution be implemented within the application,

soft requirements of the task set when run on a dedicated uniprocessor with a minimal processing speed of $f_j(m, s_j)$. When sharing a processor with additional jobs, these minimal processing speeds must be somewhat augmented to accommodate additional job time-sharing overheads, yielding the job bandwidth, $b_j(m, s_j)$, which is the subject of Chapter 4. Each job mode is also characterized by a job reward, $r_j(m, s_j)$ reflecting the job contribution to the system gain when running in mode m. As described in Section 1.3.3, jobs that can be suspended in state s_j are indicated by including a mode $m \in M_j$ such that $b_j(m, s_j) = r_j(m, s_j) = 0$.

From a meta-control perspective, the internal job state, $s_j \in S_j$, is actually hidden from it, and all the meta-controller needs to know is that an internal state transition has occurred, so that a subsequent meta-controller decision can be made. For the duration of the decision epoch, job states are assumed invariant, so we simply use M_j to denote the *currently* available mode alternatives for job, j. Similarly, the current bandwidths and rewards for each job mode, $m \in M_j$, are denoted by $b_j(m)$ and $r_j(m)$, respectively. Once the job mode has been chosen, we let $m_j \in M_j$ denote the *current* selected mode for job j, and we use the short-hand notations τ_j, δ_j, b_j, and r_j, to denote the current job task set, $\tau_j(m_j)$, the current job scheduling policy, $\delta_j(m_j)$, the current job bandwidth, $b_j(m_j)$, and the current job reward, $r_j(m_j)$, accordingly.

Job Scheduling Independence

Jobs in our program model must be *scheduler-independent* in the sense that scheduling in one job can be resolved independently of how scheduling is resolved in other jobs. This does not at all imply that tasks in one job cannot communicate with tasks in another job. After all, periodic tasks in different jobs can communicate asynchronously with each other regardless of how each task is scheduled within its own job, as long as they don't block when the message queues are empty or full. Sporadic task instances, for example, can be released by arbitrary events, including events generated by other jobs, so specific forms of synchronous communication should also be acceptable. The only essential limitation is that the timing constraints imposed by these inter-job dependencies must be formulated in a manner which enables each job to independently resolve its internal scheduling. Thus if a critical section in task A might block task B in another job, then this

anyway.

can be accounted for by ensuring that task A is characterized as not being preemptible once it enters the critical section, and by adding the critical section execution time to the task B execution time. Nevertheless, for the sake of simplicity, we will assume throughout this thesis that jobs are *loosely-coupled* in the sense that only asynchronous communication is used between jobs, even though some exceptions to this rule are readily accommodated.

Program and Platform States

The *program state* comprises all internal and environmental variables which may affect the set of jobs J, the sets of relevant job modes, $\{M_j\}$, and the set of reward and bandwidth functions, $\{r_j(m)\}$ and $\{b_j(m)\}$. The *platform state* comprises the set of platform processor capacities, $\{\beta_c\}$, which may also vary with time, e.g. when processing nodes are upgraded, when they fail, or when certain processing resources are diverted to activities outside the application.

From an RtTS perspective, the total system state, s, is therefore fully described by the combination of both program and platform states, i.e. the combination of all of the above sets: J, $\{M_j\}$, $\{r_j(m)\}$, $\{b_j(m)\}$, and $\{\beta_c\}$. Load-balancing and best-effort mode selections must therefore be reconsidered with each significant transition in s. Best-effort applications are assumed to support an adequate number of job modes and a broad enough range of mode bandwidths to best accommodate varying application and platform states.

3.1.3 Layered Software Control

The layered RtTS software control model appears in Figure 3.2. The bottom two layers facilitate the ongoing online real-time scheduling of all active tasks in the system. This is carried out independently on each processor, with the internal job scheduling being carried out by the current δ_j scheduling policy within each job, and an appropriate job time-sharing layer which decides when to dispatch the processor to each job. The decisions carried out by both of these layers are assumed to typically occur at average rates of several milliseconds, but rates here are relative.

Above these two layers, we find the job meta-control layer charged with making load-balancing decisions as well as possible mode selection decisions when jobs have more than a single mode. When dealing with single state jobs, these meta-control decisions may actually be carried out offline when

Control Layer	Level	Function	Typ. Rate
Policy Making	4	when, where (scope) & how (method & depth)	minutes
Job Meta-Control	3	mode selection & load balancing	secs
Real-Time Time-Sharing	2	job time-sharing	msecs
	1	internal job scheduling	

Figure 3.2: Layered RtTS Software Control

the application mix is static, or they may be carried out online whenever a new applications wishes to enter the system. We will usually assume the more general case where jobs have more than one state, whereupon these meta-control decisions must be carried out with each state transition in the system. We assume that such state transitions occur at rates which are much lower than the real-time scheduling rates, e.g. every few seconds, so that the internal job scheduling policies may assume that all their job tasks are perpetual.

The upper *policy making* layer must decide *when* to activate the meta-controller, for which jobs and processors (*scope*), and which meta-control policy to use (*how*). Thus, for example, the policy maker may choose to statically allocate specific jobs while other jobs are dynamically allocated, and mode-selection decisions might be carried out with each state-transition for jobs with hard deadlines, while mode-selection for other jobs might be carried out periodically at a given rate. Each meta-control policy essentially represents a specific balance between optimality and response time (*depth*), so these policy making issues will usually be determined offline as part of the system design. Nevertheless, for the sake of generality, we have included this layer as an integral part of the online control paradigm to cater to intelligent systems which might opt to carry out such decisions online.

More than any other layer, the policy-making layer illustrates that the four-tiered RtTS control model serves primarily as a framework for understanding how the various control layers must cooperate when they are needed. The number and nature of *actual* layers in any given system may vary significantly from one system to another. As we show in Section 3.3, there are applications where internal job scheduling and job time-sharing is not necessary at all, whereupon only the online meta-control layer needs to be implemented. A nice feature of the RtTS control model is that each layer can be dealt with independently, thereby promoting the development of several generic solutions for each layer. Each system can then more readily select the most appropriate suite of solutions for its needs.

Following are a few additional clarifications regarding each of these layers.

Job Time-Sharing

From the RtTS perspective, an arbitrary subset of jobs, J_c, may share any given processor, c, without *overloading* it, if the sum of current job bandwidths does not exceed the processor capacity, i.e.,

$$\sum_{j \in J_c} b_j \leq \beta_c. \tag{3.4}$$

The job bandwidths must therefore be determined in a manner which will enable the job time-sharing mechanism to facilitate the reliable execution of the internal job scheduling policies whenever Equation (3.4) holds. A major issue in job time-sharing is ensuring that lower level timing considerations within the jobs must somehow be conveyed to the job time-sharing mechanism so that they can be adequately accommodated. Another major issue is that the internal job scheduling policies be bandwidth-bound, i.e. never utilize more than the rate of machine cycles allocated to them. Job time-sharing is the subject of Chapter 4.

Job Load-Balancing

Job *load balancing* must partition J over C, i.e. produce a disjoint set of J_c subsets such that $\bigcup J_c = J$, while guaranteeing that none of the processors will be overloaded, i.e. that Equation (3.4) holds for every $c \in C$. When load balancing succeeds, each J_c is referred to as a *job cluster*, and J is considered *clusterable* if load balancing for current job bandwidths and current processor capacities is at all possible. Strongly connected jobs may also be enjoined together into job *mini-clusters* which are always allocated together. Job load-balancing is the subject of Chapter 5, and the above job *clustering* problem is shown to be equivalent to the NP-complete binpacking problem. We nevertheless find that conventional binpacking algorithms can be appropriately adapted and improved to accommodate the load-balancing needs of the RtTS architecture.

Note that the above load-balancing criteria does not include any notion of *balancing*, since the some processor utilizations might end up being significantly higher than others. Some might even argue that the common notion of *load-balancing* does not apply in the real-time context, since load-balancing is usually designed to speed-up program completion, but real-time tasks are perpetual. This is also why we will often label the above activity as *clustering*, rather than load-balancing. Nevertheless, we suggest that some notion of load-balancing should eventually be incorporated into this layer to better accommodate transient overloads when running on non-deterministic platforms, and to enable the system to initially respond to state transitions without having to carry out extensive meta-controller decisions. These issues are treated only remotely in this thesis, and the *load-balancing* label remains to encourage future work in this area.

Best-Effort Job Meta-Control

When jobs have more than a single mode, job load-balancing and job mode-selection must be carried out simultaneously. This joint activity is referred to as job meta-control, and load-balancing is therefore viewed here as a special case of job meta-control. For *best-effort* meta-control we require a system-specific criterion which we refer to as the system *gain*, v, which must be formulated as a system-specific function, g, of the current r_j job rewards, i.e.

$$v = g(\{r_j\}). \tag{3.5}$$

Whatever the gain function, g, optimal *meta-control* decisions must select job modes with those rewards and bandwidths which maximize v while ensuring that J remains clusterable. We have yet to determine what gain functions are required for typical complex real-time systems, so we have arbitrarily chosen to assume an additive gain function where

$$v = \sum_{j \in J} r_j. \tag{3.6}$$

This additive form is appropriate for our ATLAS experiment and it is also the most common assumed form in the best-effort scheduling context. This form, however, is *not* appropriate for many of the examples raised in this thesis. We contend that the notion of what is *best-effort* is very system-dependent, whereupon the most effective meta-controller algorithm must be appropriately devised for each system. What we offer, here, is an architecture which should readily accommodate a wide range of best-effort notions and algorithms. The specific algorithms devised in this thesis only come to illustrate how one of the more common and difficult notions of *best-effort* can be addressed. Exploring additional notions and seeking solutions for a wider range of gain functions is most welcome.

Job meta-control is the subject of Chapter 6, and optimal job meta-control for additive gain functions is shown to be equivalent to a composite binpacking and knapsacking problem. We have nevertheless devised two families of near-optimal meta-control policies which can adequately serve the meta-controller needs of the RtTS architecture, and which can also cater nicely to requirements for bound meta-controller response times. We also suggest how such policies can be extended to accommodate additional gain functions.

Policy Making

With optimal meta-control being intractable, each meta-control policy essentially represents a specific balance between optimality, and decision-making response times. Response times are readily profiled, but evaluating the optimality of a policy is non-trivial, and we know of no existing meta-control schemes which know how to do this. This and other related job policy-making issues are the subject of Chapter 7.

3.1.4 The Black-Box Paradigm

An important feature of the RtTS program model is that it can support the *black box job* paradigm sorely needed for systems with concurrent real-time applications. When compared with the typical adaptive control model of Section 2.2.3, we find that the while-box monitoring and performance evaluation of Figure 2.1 is replaced here by the black box requirement that each job inform the policy-maker of impending or transpired state transitions, whereupon the policy-maker decides when and how to activate the meta-controller. Though similar in nature, the RtTS philosophy is that internal job details should be hidden from the higher control layers, so it is up to the job and plaform to detect internal state transitions. This should also shorten response times and reduce online complexity. This should also be of particularly interest to systems which are required to incorporate legacy software and COTS products, since such modules are often developed independently by different vendors. Even if modules were willing to collaborate, we do not consider it practical that *all* task characteristics be publicly posted for

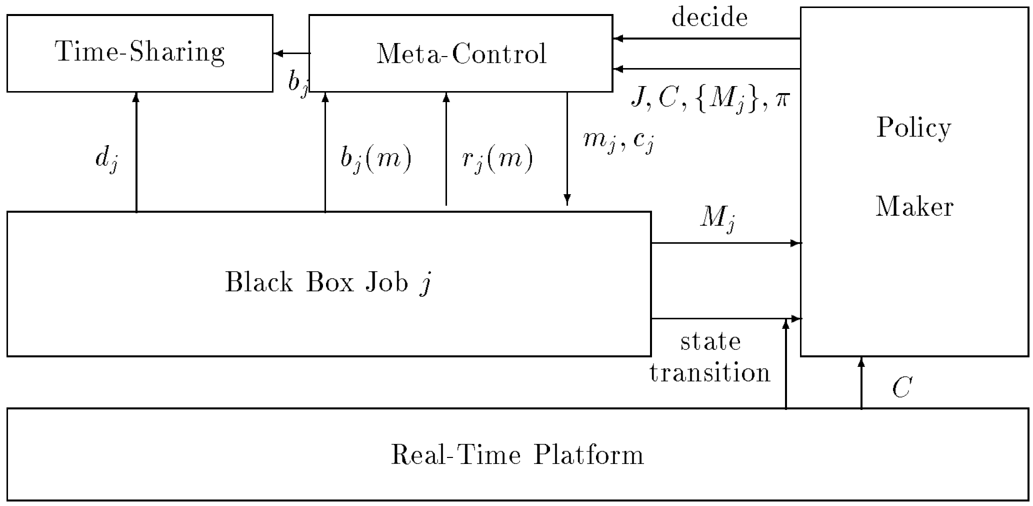

Figure 3.3: The Black Box Paradigm

an online scheduler, particularly if these characteristics are state-dependent and must be constantly updated with each state transition. And even if they could provide all this information, the dynamic resolution of real-time scheduling would be infeasible whenever joint task characteristics posed any degree of complexity.

As depicted in Figure 3.3, the RtTS framework requires very little public information from each job. As we later find in Chapter 4, the job time-sharing layer need only be provided with the job bandwidths as well as each job's current most urgent deadline, d_j, to facilitate reliable hard real-time sharing between jobs sharing any given processor. The job bandwidths are provided by the meta-controller because it is he who decides which job modes will be active. As we find in Chapter 6, the meta-control layer need only be provided with a list of jobs, J, a list of available processor capacities, C, currently relevant job modes for these jobs, $\{M_j\}$, their corresponding bandwidth and reward values, $b_j(m)$ and $r_j(m)$, the meta-control policy to be used, π, and when to decide. The meta-controller output is the current mode and processor for each job, m_j and c_j. As already noted, only the black-box job is aware of its internal state, s_j, so it must notify the policy maker whenever there is an internal state transition. It is up to the policy-maker to then decide when to activate the meta-controller and which subset of jobs and processors it should act upon.

The *black box* job paradigm defines precisely what is required from each job and how it is obtained within the RtTS architecture. As indicated by Figure 3.3, each job must notify the policy maker of all internal state transitions and it must notify the job time-sharing mechanism of every change in the job's current most urgent deadline, d_j. Each job must also provide appropriate interfaces for obtaining information regarding its current set of mode alternatives, M_j, and regarding its bandwidth and reward, $b_j(m)$ and $r_j(m)$ for each mode $m \in M_j$. Each job must also provide an appropriate mechanism for the meta-controller to alter its current mode, m_j, and determine the processor it must run on, c_j. All processors in the current RtTS model have a shared memory which contains all jobs, so that job migration should be instantaneous. But as already discussed in Section 1.3.3, job mode transitions can be quite tricky, so their timing must somehow be coordinated with the policy maker via an appropriate protocol.

3.2 Extending Portability

As already mentioned, RtTS can already be used, as is, when the immediate or extended platform family has already been established, all job characteristics are known, and all relative instruction execution times can be assumed to be equivalent for any given processor within that family. This RtTS notion of binary-level or source-level portability within a limited family of platforms should serve both hard and soft real-time applications, equally.

However, some applications are required to run on a wider range of target platforms which cannot be characterized as a family or as a small set of families. Even when the initial set of targeted platforms does constitute a family, this family may eventually become obsolete or inadequate due to technological developments or due to evolving program specifications which change with time and experience. RtTS already simplifies the maintenance of evolving subsystems, by ensuring that modifications in one subsystem will not effect other parts of the system, but porting to new platforms requires a broader notion of source-level *portability* which RtTS, alone, cannot provide. This broader notion of portability should enable us to port a top-layer application from one platform family to another by merely re-compiling, running the program through an appropriate offline task profiling process which enables each job to appropriately re-adjust its $b_j(m, s_j)$ bandwidth functions, and finally deploy the application on the new family of platforms without making a single source-level alteration within any of the black-box application jobs. Enabling us to port to arbitrary platforms, this paradigm should promote job *reuse* in our complex real-time domain and ultimately reduce the cost of software design and maintenance.

We therefore extend the RtTS architecture to support this broader notion of portability, while conceding that this extended architecture is inherently more appropriate for *soft* complex real-time applications than it is for *hard* applications. The reasons for this limitation is two-fold. Firstly, the extended architecture uses empirical task profiling techniques to estimate the task execution times and such techniques cannot generally determine accurate WCETs. Secondly, even if the task WCETs are known, current job scheduling policies might no longer be feasible on the new platform, e.g., because the adjustments in the original WCET estimates are not evenly distributed among the tasks, thereby requiring a new static CE schedule. We nevertheless believe that *hard* top-layer applications can be *a priori* designed to overcome these limitations, enabling them to port, as is, even to new platform families, and even to platforms with *non-deterministic* elements. The ATLAS experiment of Chapter 8 has been designed to explore these possibilities, and results are discussed in Chapter 9. Future research should better qualify the nature and measure of platform differences and non-deterministic elements which can be accommodated in a hard real-time context, and appropriate extensions should expand the scope of catered applications. As previously suggested regarding extended families in Section 3.1.1, such platforms should ultimately be characterized by a set of capacities designed to serve appropriate mixtures of hard and soft tasks. In the meanwhile, however, we can recommend the extended architecture for only complex real-time applications which are soft.

Enhancing Portability

Porting to yet unknown and possibly non-deterministic runtime platforms requires several additional mechanisms which the kernel RtTS architecture does not provide. First we require that internal job design and scheduling be resolved in a platform-independent manner, totally disregarding the hardware and software architectures within the underlying platform layers. Note that this is not essential when targeted for a specific platform family. Secondly, we require appropriate offline mechanisms for profiling application behavior on a new platform so that job bandwidths can be appropriately

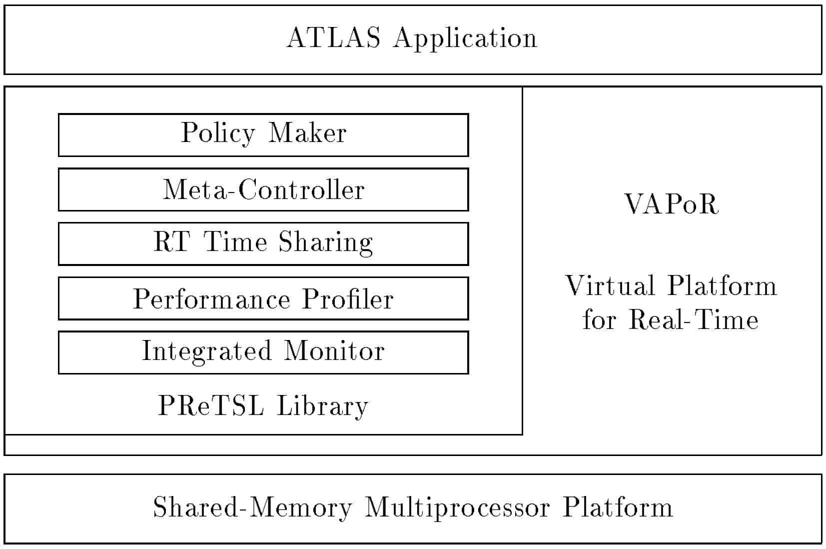

Figure 3.4: The Top Layer Architecture

adjusted. Thirdly, we realize that empirical task profiling is insufficient when seeking task WCETs, primarily when dealing with non-deterministic platforms, so we desire appropriate mechanisms for anticipating processor overloads and for avoiding them by appropriate online adaptation.

To accomplish this we have extended the basic RtTS architecture to obtain the top-layer architecture of Figure 3.4. Familiar elements from the RtTS architecture include real-time *time-sharing*, online *meta-control*, and an optional online *policy maker*. New elements include the integrated monitor, the performance profiler, the PReTSL library, and the virtual real-time platform, VaPoR. The minimally-intrusive *integrated monitor* evaluates platform-dependent characteristics, anticipates faults, and verifies performance. The *performance profiler* uses monitor-provided information to compute job bandwidths for all states and modes. The VaPoR virtual platform ensures that all platform services are obtained in a platform-independent manner. And the portable PReTSL real-time library attempts to provide all the necessary RtTS and top-layer control mechanisms in a generic, accessible, and platform-independent format.

The VaPoR Virtual Platform

VaPoR is designed to serve as a generic *virtual platform for real-time* applications, and must therefore support a comprehensive set of operating system services typically required by such applications. As in Chaos [SGZ90], ARTS [TM89], RT Mach [TNR90] and other systems [MS92], a major VaPoR premise is that real-time applications consists of *cooperating* processes which enable them to use light-weight threads to implement concurrent tasks. Implementing job tasks as user-managed threads enables us to run such applications on a wide range of conventional platforms which do not provide such services at the kernel level. Implementing the RtTS control layers at the user-level also enables us to more readily customize them by incorporate only those layers and those algorithms which are needed for each system. VaPoR, on the other hand, will know how to exploit such kernel-level services when available. On a RT Mach platform, for example, job tasks would be translated by VaPoR into native Mach threads. On a Java virtual machine, job tasks would be translated to Java threads. On a non-threaded UNIX platform, VaPoR might incorporate C or POSIX threads for basic timing, thread synchronization, and message-passing. VaPoR provides a uniform interface to these services to maximize portability. As in MOSIX [BGW91], and other message-passing environments [McB94], VaPoR must be designed to efficiently and reliably deliver messages between node-independent jobs regardless of their current placements. This should be quite straight-forward when catering to shared-memory platforms. VaPoR abstractions avoid dictating the mechanisms to be used by these services so that the current platform architecture can be best exploited[2].

Unlike most other experimental real-time platforms, the VaPoR approach attempts to accommodate widely accepted industry standards, rather than attempt to supersede them. VaPoR adopts an Ada-like strategy whereby various services are either internally resolved within user-level run-time libraries or leased out to the platform operating system in a manner transparent to the application. It also provides all the necessary hooks to the integrated monitor and profiler so that program and platform behavior can be monitored and profiled.

Not all tasks on a given platform will necessarily be cooperative. A conventional X server, for example, will typically comprise of heavy-weight

[2]Mach 3.0 with a Unix server was probably the first to demonstrate that a portable virtual platform can outperform even a dedicated non-portable platform (Ultrix 4.0) by using appropriate platform abstractions and by optimizing each instantiation [CB93].

tasks with higher preemption overheads, and its bandwidth will depend on the events it receives. It will enforce no self-restraint, so it must be assigned a lower priority. We can then make sure that it receives an adequate bandwidth by controlling the available platform capacity and by ensuring that all application jobs consume only the bandwidths allocated to them.

To maximize the portability of the top-layer application, all platform services are obtained via the VaPoR layer, whereupon only the VaPoR layer must be re-implemented and customized when extending the current set of supported run-time platforms.

The PReTSL Library

All portable contingents of the RtTS control layers and top-layer extensions are contained in a portable real-time support library, PReTSL, thereby maximizing the platform-independence of the architectural elements. Non-portable elements are isolated within the VaPoR layer which mediates between the application and the platform.

Integrated Monitor

The integrated monitoring subsystem plays a very central role in a top-layer architecture, primarily serving system validation, performance analysis, performance profiling, and job scheduling. The integrated monitor relies on a dynamic combination of minimal-intrusive *probes* and *sensors* embedded within the VaPoR, Pretsl, and program layers to sample variables and trace execution. A job scheduler using *milestoning* [HS89], for example, relies on implanted milestone sensors, which monitor task progress to appropriately reduce the WCET assumed for that task for more efficient online scheduling. All monitored data must be time-tagged, collected, filtered, routed, processed, and recorded, while adapting to the dynamic requirements and resources of the system. Integrated monitoring schemes have already been devised for several dynamic and distributed real-time systems, e.g. [OSS90].

Performance Profiler

As illustrated by the Concord project and the Flex programming language [KL91], safe WCET estimates can be produced and perfected by combining compile-produced data with accumulated run-time experience. Most profiling activities are carried out offline, to minimize the run-time overheads. The Concord project also shows how calibrated WCET functions can be obtained which provide tight WCET upper bounds for each task as a function of predetermined state variables, so that job mode bandwidths can be upgraded with each state transition. Similar techniques are currently being incorporated into PERC, letting the administrator of each applet use profiler-based execution time estimates to negotiate the applet requirements for a given platform with the run-time kernel. In a similar fashion, the *performance profiler* in our top-layer architecture is charged with providing the platform-dependent task attributes, e.g. typical or worst-case task execution times, so that each job can appropriately adjust its mode-dependent and state-dependent bandwidth functions.

PERC profiling also intends to take into consideration underlying platform activities such as dynamic memory garbage collection and caching overheads introduced by task-switching. In a top-layer context, platform activities which are not directly associated with specific job activities are collectively referred to as *platform jobs*, and the performance profiler must monitor their bandwidth behavior so that available platform capacities can be appropriately reduced to accommodate them. When dealing with non-deterministic platform activities, we will refer to them as platform *noise*, also modeled as an independent platform job with a bandwidth reduced from the platform capacity. As discussed in Chapter 9, platform noise can thus be accounted for without jeopardizing the schedulability of the application jobs as long as the noise level does not exceed the allocated bandwidth. The actual noise bandwidth should therefore be monitored by the online profiler so that it can warn the online policy-maker of a rising noise bandwidth which might exceed the allocated bandwidth. The policy-maker can then decide to increase the allocated noise bandwidth and activate the meta-controller to best-effort adapt to the new processor capacities. Such techniques should eventually enable appropriately-designed *hard* real-time applications to effectively accommodate new platforms and non-deterministic platform elements as well.

3.3 Sample Applications

The purpose of this section is to show how several examples of realistically complex real-time systems might benefit from the RtTS and top-layer architectures.

CPUs	I/O		
	Disk	$\Longleftrightarrow$	CD/DCD
Active	Video	$\Longleftrightarrow$	TVs
Job	MIDI	$\Longleftarrow$	Keyboard
Set	Audio	$\Longleftrightarrow$	Mic/Spkrs
	Fax/Modem	$\Longleftrightarrow$	Phone Lines

Figure 3.5: An Integrated Multimedia System

3.3.1 A Concurrent Multimedia System

To appreciate the power of real-time concurrency, consider the case of a general purpose multimedia system, targeted for a family of user platforms, providing a variety of *concurrent*, user-selectable services, e.g., interactive video, musical playback and synthesis, fax and voice mail, electronic shopping and banking, and various internet services. Each platform may have a different number of shared-memory family processors with various capacities, and, as depicted in Figure 3.5, each platform may be equipped with a *customizable* set of I/O devices, including any number of CD players for audio and video playback, any number of video outlets for displaying interactive video, a MIDI keyboard for playing synthesized music via any number of audio speakers, microphone inlets for recording and mixing music, and any number of phone lines for voice mail, Fax services, internet access, and computer-run video conferencing. Most of these above-mentioned multimedia services are real-time applications, many of them having hard deadlines. The playing of synthesized music, for example, might be considered *hard* real-time, because random lapses in the generated music will not be tolerated, else no one will buy the system. When generating video frames, on the other hand, a few missing frames might not be noticed, so some aspects of video playback and conferencing might be considered *soft*. And besides all these multimedia-related services, the same system can concurrently provide regular office-related services, such as word processing, scheduling, and electronic mail. Such office-related services are generally treated as non-real-time, but poor response times can be quite annoying and should be avoided if possible.

The number and nature of concurrent services is dynamic and dictated by the user's needs. The user might request that a given CD be piped into the bedroom at a specific time, and that a given Fax be sent out a few minutes later, while the user prepares a document on the word processor, the kids enjoy an interactive video in the den, and an internet agent goes surfing to seek the best investments on the stock market. It would be very undesirable if the CD suddenly stuttered or the word-processor ceased to respond when the Fax went on or when the kids did a fast-forward. What we would like to see in such systems, is that we should be able to dictate a minimal quality of service for each of these services, and the system should refuse to carry out a service if these minimal qualities cannot be met. Thus the CD or Fax would not go on if this would disrupt the other, already active, services.

But how do we evaluate the schedulability of an arbitrary set of services on an arbitrary platform in the family? Note that many of these services require several non-independent tasks, some sporadic, some hard. In the video playback service, for example, one task would have to control and read the DCD, a second task would have to decompress the MPEG-coded video frames, a third task would have to feed the video outlet, and a fourth task would have to monitor the video controls and control the other tasks accordingly. The output of one task is the input to another, so the first three tasks are not independent and access to their shared data must be synchronized. DCD access times and MPEG decompression times may vary from frame to frame, so task execution times are not fixed, but a missed frame might not be noticed. The untimely feeding of the video outlet, on the other hand, would be intolerable, and the video control monitor must respond quickly enough to ensure that a video pause will indeed capture the intended frame. Scheduling tasks for a given set of services might be quite simple when targeted for a specific processor capacity. But scheduling, here, must be carried out *online* for arbitrary processors and capacities within the platform family and for arbitrary sets of tasks belonging to arbitrary sets of concurrent services.

As indicated by Table 3.1, to further complicate matters, all of the above services have different states, and required resources are state-dependent. Fast forwarding a video, for example, would require a much higher DCD access rate, making this state significantly more demanding than playback at normal speed. Does this mean that we must always assume task execution times for the most demanding service states even if those states are relatively rare? And if worst-case requirements are ten times greater than typical requirements, do we essentially run our system at one tenth of its capacity?

In a top-layer design approach, each system service would be provided by an independent *job*, each job supporting several alternative *modes* of

job	states	modes
interactive video	play, FF, pause	frame rates, resolutions
fax & voice mail	tea, dial, Rx, Tx	baud rates, resolutions
elec. shopping/banking	"	baud rates, text/graphics
MIDI Synthesis	MIDI-dictated,	sample rates, synthesis methods
prog. CD Player	play, pause	filters
internet agent	sleep, query, analyze	response times
misc. tools: e.g. editor		response times

Table 3.1: Multimedia States and Modes

operation, as listed in Table 3.1. Each mode represents a different quality of service for that job, each requiring a different bandwidth of computational resources for each job state. Internal scheduling for each job mode and state is resolved offline and independently, whereupon the schedulability of an arbitrary set of jobs is readily determined by attempting to load-balance the bandwidths for the worst-case states of the minimal acceptable modes. Once accepted, online meta-control will always select the highest quality modes possible for the current job states, so that most services will generally provide high quality services, making occasional lower-quality services more tolerable. Job states do indeed change at reasonably low rates, and the response times to user requests for new services can be quite tolerable, so that these occasional meta-control overheads are negligible. The rewards contributed by each job are readily customized to reflect user preferences. Thus when fast-forwarding the video, the preferences might be such that the video quality will be reduced rather than infringing on any of the other services. To accommodate an urgent fax transmission, the internet agent might be provided with a suspend mode, thereby causing it to be suspended until the transmission is complete. When editing a very urgent document, a larger bandwidth would be allocated to the word-processing service at the expense of other services.

Note that in a conventional real-time scheduling scheme, non-real-time activities will always receive the lowest priorities, causing them to starve when real-time activities have what to do. The RtTS time-sharing scheme enables us to better manage the available resources in a manner which enables us to limit the bandwidths absorbed by the real-time activities so that non-real-time services can benefit as well. And a smart editor would provide a state-dependent bandwidth function which would prescribe the bandwidth necessary to complete the current editor operation within a mode-dependent response time.

Using a conventional task-oriented approach, real-time scheduling and load-balancing would be horribly complex, since the set of concurrent jobs is user dependent, available computational resources are platform dependent, and job task requirements are state, mode, and platform-dependent. The encapsulated scheduling solutions in our job-oriented top-layer approach enable us to focus primarily on the issues of load-balancing and best-effort online adaptation. In fractions of a second, each user service request is instantly tested to determine whether it can run concurrently with already running jobs, after having considering all mode-transition and load balancing alternatives. The ability to improve overall system services is re-evaluated with each internal job state transition. And the black-box paradigm enables us to develop and maintain each of these services independently without having to divulge internal task characteristics which can be arbitrarily complex.

3.3.2 A Missile Simulator

In the concurrent multimedia example, the top-layer architecture helped cater to the needs of an arbitrary set of concurrent applications with complex task characteristics. The following HIL (hardware-in-the-loop) missile simulator illustrates the benefits of a top-layer architecture when application requirements vary significantly from one moment to the next, even when dealing with a *single* application with simple task characteristics. A typical requirement with such simulators is that they run on an *a priori* known platform, but that platform will eventually evolve through different numbers of processors and capacities, thereby constituting a family of platforms. Such portability requirements are typical for expensive platforms which will eventually be expanded and upgraded with time.

The number of supported simulation scenarios is virtually unlimited, each scenario comprised of an arbitrary set of interacting entities, e.g. internal missile components, a variety of targets, and electronic counter-measure devices, with predetermined modes of behavior. Simulation is carried out in steps, with the current state of each entity being modeled as a function of entity state vectors from previous steps. The current state is then fed to appropriate hardware in a timely manner at the end of each step to close the loop in the HIL simulation.

When running on a shared-memory multiprocessor, each processor can handle a different set of entities. All activities are periodic with identical periods equal to the simulation step size, so real-time scheduling is quite trivial. The more difficult question is how to make the most of the available platform resources to maximize the simulation precision. Thus, the system *gain*, here, is simulation precision. The simulation step size is usually fixed and dictated by the hardware elements in the simulator. What *can* vary is the simulation model used to model the behavior of each entity, each model having different computational requirements. Letting each entity be represented by a job, each simulation model would be a job mode, each mode contributing a different reward to the overall simulation precision, and each mode requiring a different computational bandwidth.

Mode contributions, however, are very state dependent, since the advantages of one model over another will often depend upon the distances, the angular velocity, the orientation, etc. Thus the optimal set of job modes must constantly be re-evaluated, online, as a function of the current simulation state. This is precisely what the RtTS architecture is designed to accomplish, and we know of no other approach which enables us to do this. PERC, for example, is simply not equipped to renegotiate resource allocations, with each state transition, in real-time.

Note that there is no need for encapsulated job scheduling and for real-time job time-sharing in this example, since new states for each entity are computed in an integrated manner for each simulation step. Thus, the entity-related computations can be carried out sequentially at each step for all entities allocated to each processor. With the entity state vectors in a shared memory, each new step can be carried out with a new set of models and a new allocation of entities to processors. Each step can therefore entail also a new meta-controller decision by simply treating the meta-controller as an additional job. All meta-controller decisions guarantee that the sum of mode bandwidths will not overload the available platform capacities, so that timely step-wise control of the simulator hardware is also *hard* guaranteed.

3.3.3 A Virtual Reality Application

A virtual reality system enables users to immerse themselves in a virtual world which responds to the user's behavior in real-time. As in most real-time systems, the user interface to the virtual world must provide sensors for sensing the virtual world as well as actuators for acting upon it. In many systems, an appropriate head set provides the seeing and hearing sensors for perceiving the virtual world, and joystick-like actuators enable the user to move within the virtual world and carry out predefined actions. Whatever the human interface, a virtual reality system must monitor the actuator states, must simulate the interactive behavior of all entities within the virtual world, and must control the sensors to emulate the current world state. When *visual* sensors are supported, the process of creating the perceived images is called *rendering*. A precise real-time simulation and rendering of any virtual world is rarely feasible even on the most powerful existing platform. So the primary virtual reality objective is to choose the entity simulation models and rendering techniques which will maximize the overall *realism* produced by the system given a limited amount of available platform resources. As in the missile simulator, each simulated entity can be treated as a job with several modes, but there is no hardware in the loop (only a human), so each mode may now adopt a different state update rate as well as a different simulation model, and most deadlines are now soft. Thus each job mode may constitute a totally different set of tasks with different timing constraints, so that scheduling and schedulability is no longer trivial. As in the missile simulator, the realism contributed by each entity's mode is often a factor of the entity's state, e.g. the nature of its motion as well as its observed size and visibility. Thus a best-effort choice of simulation models and rendering techniques must also consider the current state of all world entities, as well as the schedulability of each possible choice.

Even when targeted for a specific platform, we know of no alternate approach which can handle such complexities. As in the missile simulator, each entity is simulated by a separate job, and all sensor and actuator activities are also modeled as jobs. Due to the more complex nature of these tasks, task scheduling is resolved offline independently for each job mode and each job state. Moreover, sensor and actuator activities may be scheduled using policies appropriate for hard deadlines, while the simulation activities can use more speculative policies which are better suited for soft deadlines. The state-dependent job mode rewards and the system gain function must be designed to reflect the contribution of each job mode to the overall realism of the system. The best-effort meta-controller will then determine which set of job modes will maximize the realism produced by the system for any given state, whereupon jobs are load-balanced accordingly. Jobs sharing a given processor will then use an appropriate real-time time-sharing mechanism to interleave the internal job schedules already predetermined for the current job modes. As already discussed in Section 1.3.3, all we have to ensure is that the meta-control adaptations are carried out in a timely manner.

And once designed, the virtual reality system will readily port to additional platforms as well.

3.3.4 A Cellular Base Station

A cellular phone base station serves as a conduit between a conventional public phone system and a limited number of wireless subscribers within a designated area covered by the base station. It is in the best interest of any base station to maximize the number of concurrent subscribers it can support without sacrificing the quality of the connections. Quality service is obtained by employing a variety of voice enhancement and noise-cancellation techniques, but the computational resources available to each base station is limited and may vary from time to time as processors are added, removed, or upgraded. A disruption of services is not to be tolerated, so such systems are expected to automatically adapt to platform reconfiguration on-the-fly.

Connection noise levels and the degree of subscriber cross-interference is also very dynamic, so that noise-cancellation and voice enhancement techniques must be chosen in accordance with the current connection state and the desired quality of service. Ultimately, what we need is a best-effort base-station which allocates resources to each connection in accordance with its current state, while attempting to maintain a *uniform* quality of service for all subscribers, rejecting incoming subscribers only when this uniform quality of service will descend below a minimal level. Each connection is therefore readily modeled as a job with several possible connection states and several alternate job modes, each mode using different set of techniques and parameters to enhance the quality of service, thereby contributing a state-dependent reward to the overall quality of service. An appropriate system gain function must be devised so that the selected job modes will provide a uniform quality of service, and the meta-controller decisions must be re-evaluated with each new connection or disconnection, with each altered platform configuration, and with every state-transition in any of the connections.

3.3.5 A Remote-Control Vehicle

Jobs in our previous examples generally entailed a very small number of tasks, sometimes even a single task. In large scale systems, however, we suggest that each job might constitute a complex subsystem on its own. The remote-controlled vehicle of Figure 3.3.5 would be such a system. The *video*

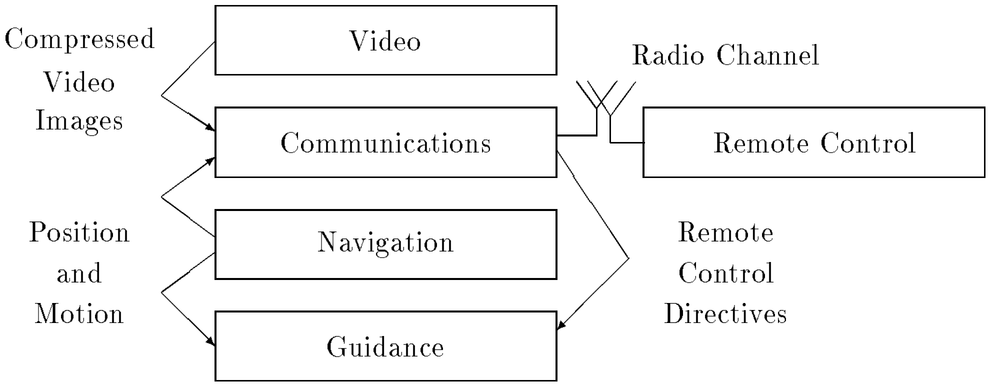

Figure 3.6: A Remote-Controlled Vehicle

job, in this example, is responsible for video acquisition and compression, which can be carried out at various frame rates and resolutions, using any of several compression algorithms. The *navigation* job periodically determines the vehicle's current position and motion at appropriate rates. The *communication* job transfers video and navigational information to the remote control system via radio, while catering to various interference levels by using appropriate redundancy and error correction techniques. The *guidance* job controls locomotion actuators to comply with remote control directives at various speeds. Rates, resolutions, levels, speeds, and techniques are all dictated by any number of selectable modes per job. Each job mode is assigned a *reward* which reflects the job priority and the benefit of using that mode *vis a vis* other job modes, e.g. a 30 frames per second mode might have a reward which is 50 percent better than that of a 10 frames per second mode to quantify the difference in video quality. The mode *bandwidth* prescribes the collective computational requirements of that mode. Mode bandwidths are often *state*-dependent, e.g. in an inertial navigation subsystem, the rate of navigational positioning must be higher when the craft travels faster, thereby requiring a greater bandwidth. In an autopilot mode, self-navigation requires even greater bandwidths, and the complexity of self-navigation is very much a function of circumstances. In a similar fashion, dynamic rewards may be used to reflect varying job priorities under various circumstances, e.g. navigation is more important than video once a final destination has been determined. Active job sets may differ for different phases of a mission, e.g. during takeoff and landing. Runtime *platform* characteristics may also vary with time, as when processing clusters are upgraded, when they fail, or when certain cluster resources are diverted to other activities.

Unlike some of the previous examples, even a temporal failure of any of these subsystems might jeopardize the mission, and the remote-controlled system must be capable of overcoming the many possible difficulties on its own. Considering all possible circumstances for such a system and preprogramming it to deal with every single circumstance would be futile. Thus an automated best-effort design would appear to be the most appropriate approach for such a system.

3.3.6 Inherent Strengths and Weaknesses

The above examples illustrate that the RtTS architecture is readily applied to a variety of complex real-time applications. As illustrated by the multimedia, simulator and base station examples, even the somewhat restrictive notion of RtTS portability within a specific *family* of platforms is meaningful for many systems, and the top-layer extensions of Section 3.2 can already extend this notion of portability for softer applications such as the virtual reality system.

Managing Complexity

The common denominator in all of the above systems is that the number and nature of activities vary with time, and catering to their worst-case requirements would require a prohibitively large-scale and expensive platform. To accommodate these activities on a smaller and less expensive platform, each activity supports several different modes of operation, each mode representing a different tradeoff between the required bandwidth of resources and the contribution of that activity to some notion of a system gain. Choosing the best set of tradeoffs for a given platform would best be carried out offline if not for the state-dependence of the required mode bandwidths and the large variances in bandwidth from state to state. Catering to the bandwidths of the worst-case state for each mode might result in a very low platform utilization and a very poor system gain. The primary strength of the RtTS architecture is that it provides a framework for *online* adaptation to each state to maximize the system gain over time for any given platform. Decomposing an application into *jobs* allows us to independently analyze the behavior and contribution of each activity or sets of activities. As illustrated by the cellular base station example, such a system can also adapt on-the-fly to changes in the platform configuration, making it appropriate for fault-tolerant systems as well. We know of no other approach which can effectively accommodate the complex needs of such systems, and we suggest that the lack of alternatives is the primary reason why such adaptable applications are not prevalent.

Architectural Flexibility

Another inherent strength of the RtTS architecture is its flexibility, making it applicable to a wide range of hard, soft, and even non-real-time applications in which response times are important, as in the editor in the multimedia example. Job granularity is fluid, ranging from the single task in the missile simulator to a complete subsystem in the remote-controlled vehicle. The job *mode* abstraction introduces arbitrary degrees of flexibility regarding how jobs can be accomplished. In any of the above examples, secondary support activities such as file servers, schedulers, recovery mechanisms, and even the meta-controller itself would also be classified and treated as jobs, with possibly more than one mode per job, each providing a different level of support to best accommodate available resources while also considering the bandwidth needs of other system components, as well. Underlying platform activities may also be modeled as jobs, as with the platform noise of Section 3.2.

As illustrated by the above examples, the RtTS and top-layer architectures are essentially *frameworks* for designing platform-independent complex systems, and not all elements are always necessary. In many applications, only a few of the RtTS control layers need be implemented, e.g. only the meta-controller layer in the missile simulator. In every application they can be implemented in a different manner, e.g. a prioritized best-effort in the multimedia example and an equalizing best-effort in the simulators and base station. The state-dependent bandwidth and reward functions are also flexible, job states possibly also considering recent bandwidth allocations as a feedback mechanism to ensure that rewards are gradually raised for jobs which have not received enough meta-controller attention. Thus job mode rewards can represent job contributions to a carefully crafted system gain function, or they might represent a more conventional notion of priority.

Scheduling Flexibility

The choice of scheduling solutions for the application is also very flexible, being independent for each job mode, and also independent of the meta-control policies being used. The independent job mode scheduling solution can be deterministic or speculative, static or dynamic. Common *deterministic* schedulers for *hard* real-time task sets include the *static* CE, and the *dynamic* RM and EDF policies of Section 2.1. When dealing with *soft* real-time tasks, we might prefer more *speculative* or best-effort scheduling policies, e.g. those of Locke [Loc86] and Miller [Mil90], which can provide higher value-based performance. We might also want to use the job abstraction to isolate *hard* tasks from soft tasks. Thus a single plug-in software module in the multimedia system might actually be designed as a *mini-cluster* of jobs, i.e. strongly connected jobs which are always allocated together. Each job in the mini-cluster could then have a different scheduling policy, whereupon each task would be allocated to the job which has the most fitting scheduling policy for it. Jobs communicate with each other via adequately buffered asynchronous mechanisms so that scheduling within one job will not affect scheduling in the others. Note that a single task also qualifies as a job, whereupon it need not be scheduled by an internal δ_j scheduler, since it could essentially be scheduled by the real-time time-sharing control layer.

New Crucial Problems

These examples also illustrate that such an architecture introduces several new crucial issues which have to be resolved, e.g., how to decompose applications into jobs, how to define *best-effort*, how to devise appropriate system gain and job reward functions, how to choose algorithms with reasonable overheads for each control layer, how to get all of the architectural elements to work together, and how to best implement these mechanisms on conventional platforms. All of these issues can be resolved *offline*. But clearly, seeking comprehensive top-layer solutions for a wide range of complex real-time systems can be a very expensive inter-disciplinary endeavor. Full-fledged top-layer architectures are therefore not recommended for relatively simple real-time applications which have viable conventional alternatives. A nice feature of the layered architecture is that various aspects can be incorporated selectively to accommodate various levels of system complexity. Addressing all of these crucial issues is beyond the scope of this thesis, so we have therefore chosen, in the remaining chapters, to deal with a sufficient subset of ground-breaking issues necessary to show how such a top-layered system might be implemented. We then show by example how the wider scope of issues can be resolved for a specific application.

3.4 Conclusions

A primary difficulty in designing, implementing, and testing *complex* real-time systems is the sheer complexity of making software control decisions at various levels. Real-time scheduling on each processor, load-balancing between processors, online selection of optimal software adaptations, each of these issues is independently complex. Strong dependencies between these control layers make matters considerably worse.

The primary objective of the RtTS software architecture is to facilitate the orderly construction of a complex real-time system in a manner which will considerably reduce solution complexity and significantly enhance portability and concurrency. Independently scheduled jobs and real-time time-sharing reduce the complexity of online real-time scheduling for dynamic sets of tasks from concurrent applications. The bandwidth abstraction simplifies load-balancing and best-effort online adaptations. It also supports a black box paradigm sorely needed for concurrency and for incorporating legacy software and COTS products. Top-layer extensions to this architecture enhance the portability of such applications. The architecture is also flexible and not overly restrictive.

Chapter 4

Real-Time Time-Sharing

This chapter discusses one of the most essential ingredients of our top-layer approach: job *time-sharing* for an arbitrary job cluster, J_c, on a processor, c, with capacity of β_c. This job time-sharing problem has to be resolved *online* because the set of jobs, J_c, is not *a priori* known. As already described in previous chapters, we assume that the job state, s_j, for each job, j, varies only occasionally, so that the internal job schedulers and the job time-sharing mechanism may assume that the current state will persist long enough to be treated as perpetual. From a job time-sharing perspective, then, the current job mode, m_j, for each job, j, and the current job state, s_j, are fixed, so that the current job task set, τ_j, and the current internal job scheduling policy, δ_j, are also fixed. We can therefore assume the simpler static program model of Section 1.3.2 throughout this chapter.

The real-time time-sharing problem, therefore, is to define a job bandwidth, b_j, and devise a job time-sharing algorithm which will always guarantee the temporal requirements of all tasks for all jobs, $j \in J_c$, if the sum of bandwidths does not exceed the processor capacity, i.e.,

$$\sum_{j \in J_c} b_j \leq \beta_c. \tag{4.1}$$

This schedulability criteria is conservative, but this is the maximum that we believe can be attained with black-box jobs, and we will also show that this can be near-optimal when dealing with well-designed jobs.

Let f_j be the minimal processor capacity needed to guarantee the temporal requirements of τ_j when using δ_j on a dedicated uniprocessor[1]. Clearly,

[1]Note that the τ_j task set and task characteristics, e.g. task execution times and

the job bandwidth, b_j, must be somewhat greater than f_j since it must accommodate worst-case job time-sharing overheads as well. The job time-sharing algorithm, on the other hand, must minimize the time-sharing overheads and must ensure that b_j can be determined independently for each job so that Equation (4.1) can serve as a schedulability criteria for *arbitrary* J_c clusters. For simplicity, we assume that the job task sets are such that they are readily accommodated by scheduling policies surveyed in Section 2.1, e.g. CE, RM, and EDF, possibly augmented by a priority management protocol such as PCP (Section 2.2.1). We also assume that these have been implemented in a *bandwidth-bound* manner, i.e. that they will never consume more than the allocated rate of machine cycles, b_j.

As already emphasized in the previous chapter, the two-tiered real-time time-sharing scheme of Figure 3.2 is a framework which can be applied differently to each system. In the missile simulator example of Section 3.3.2, all the real-time tasks in the system are periodic with the same period, all being equal to the simulation step size. In such a system, all tasks in J_c can be executed sequentially in an arbitrary order for each simulation step, so that there is no need for internal job scheduling, and job time-sharing is trivial. If all jobs in the system have static CE scheduling solutions, and all CE solutions have the same cycle time, then job time-sharing is quite simply accomplished by concatenating the CE schedules for J_c on each processor, c. If all CE solutions have different cycle times, we need a somewhat more sophisticated job time-sharing scheme, e.g. dynamically interleaving the CE schedules on an EDF basis. Such a solution would be more than adequate for most of the sample systems surveyed in Section 3.3. If different scheduling policies are being used for different jobs, and if some of the job tasks are sporadic, we need even more sophisticated job time-sharing solutions. When dealing with jobs with *hard* deadlines, we require job time-sharing solutions which can always provide deterministic guarantees for all hard deadlines on each processor, subject to Equation (4.1).

To this end, this chapter explores the more general and complex case where each job may have both periodic and sporadic tasks, an arbitrary percentage of these tasks may be hard, and the jobs in J_c might be using different bandwidth-bound scheduling policies to address their internal needs. Such a diverse mixture of jobs would be likely in the concurrent

maximum blocking times, are *a priori* known as a function of m_j and s_j, so there is always some minimal f_j capacity and some δ_j policy which will satisfy these temporal needs.

j τ_j	Job 1 Task A	Job 2 Task B	
release, R_i	$30i$	$50i$	msecs
deadline, D_i	$30i + 10$	$50i + 10$	msecs
computation, n_i	1000	1000	machine cycles
bandwidth, b_j	110	110	MHz

Table 4.1: A Simple Single-Task Per Job Example

multimedia example, in which each service is developed independently. To address the needs of such systems, Section 4.1 presents the EDF job time-sharing scheme, followed by a detailed example in Section 4.2. The primary issue is to show how low-level timing considerations with the jobs are taken into account by the higher level time-sharing mechanism. We then suggest in Section 4.3 how jobs might be designed to minimize these overheads so that EDF time-sharing might even be near-optimal. Conclusions are provided by Section 4.4.

4.1 EDF Job Time-Sharing

As already mentioned in Section 2.1.5, EDF scheduling is a classic real-time scheduling policy capable of maximizing processor utilization. But EDF overhead is more significant than its RM counterpart, since task priorities must be updated with each release of a new task instant. A significant advantage of EDF scheduling is that the number of task context switches is upper bound by the twice the number of task releases. Thus the overheads of EDF scheduling at the job level might be significant, but they are limited by known factors so that they can be fully taken into consideration when dealing with hard deadlines which must be guaranteed.

This section uses a simple example to introduce the EDF job time-sharing scheme in Section 4.1.1, shows how it might be implemented in Section 4.1.2, qualifies the nature of supported internal job scheduling policies in Section 4.1.3, and proves that such a scheme can provide hard guarantees in Section 4.1.4. A more detailed example then follows in Section 4.2.

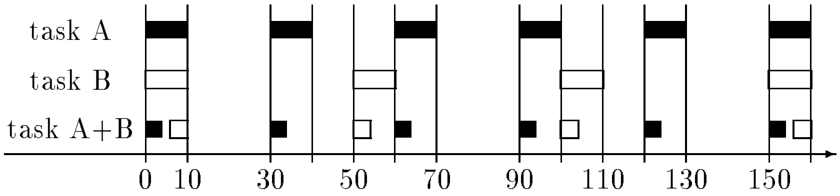

Figure 4.1: EDF Scheduling for the Simple Example

4.1.1 A Simple Example

Consider the simple example of the two periodic tasks of Table 4.1, Task A and Task B, both released at fixed intervals but required to complete within 10 milliseconds. Task A is released every 30 milliseconds, hence the task A release times, R_i, occur at times $30i$, and the task A deadlines, D_i, are at times $30i + 10$. Task B releases occur every 50 milliseconds. Treating each task as an independent job, we now resolve the issue of internal job scheduling independently. In a single task job, scheduling is trivial, just execute the task as soon as it is ready.

So all we really need now is to determine the job bandwidths. As previously described, to determine the job bandwidth we must first determine the minimal processor capacity required to satisfy the temporal requirements of each job when running on a dedicated uniprocessor. In this simple example, the minimal processor capacity is clearly that which will enable each task to complete in 10 milliseconds. Using Equation (3.2), Given that the number of machine cycles, n_i, required by each task, i, is 1000 machine cycles, then the minimal processing capacity must be at least 100 MHz. But some control element must ensure that the tasks are released on time, so we also take into account that there is a 200 Hz system clock handler which releases these tasks with an additional worst-case overhead of 50 machine cycles every 5 milliseconds. This adds an additional bandwidth of 10 MHz, so the actual minimal processing capacity is 110 MHz. The same timing mechanism can also serve both jobs when sharing a given processor, so job time-sharing overheads are zero, i.e. the job bandwidth, b_j, for both jobs is also equal to $f_j = 110$ MHz. According to Equation (4.1), both jobs should therefore run reliably on a shared processor with a 220 MHz capacity, whereupon, according to Equation (3.2), each task will complete within $1000/220 = 4.55$ milliseconds and the system clock handling will get activated every 5 mil-

```
procedure UpdateEarliestJobDeadline( j, d_j );
/* Job j notifies that its closest deadline is d_j */
begin
        /* Find the currently executing Job j' and its deadline d': */
        Let (j', d') = first element in JobDeadlines ;
        Update j's entry in JobDeadlines ;
        if d_j < d' and j ≠ j'
          then preempt j' and dispatch j;
          /* then the scheduler of job j will now take over. */
end;
procedure EventsHandler( j, event_type );
begin
        Execute the relevant event handler of Job j;
        /* which might call upon UpdateEarliestJobDeadline */
end;
```

Figure 4.2: Updating Earliest Job Deadlines

liseconds and complete within $50/220 = 0.45$ milliseconds. As illustrated by the combined scenario in Figure 4.1, the deadlines of both jobs will always be met. Due to the trivial nature of the jobs in this example, job time-sharing was accomplished with *zero* job time-sharing overhead.

4.1.2 EDF Time-Sharing Implementation

In a more general EDF job time-sharing scheme, we must consider all the necessary mechanisms which are needed to support such a scheme, whereupon the job bandwidths must be augmented to accommodate the worst-case overheads of there mechanisms. One possible set of EDF support mechanisms is presented in this subsection. Another alternative implementation appears in Section 4.1.4.

An EDF job time-sharing mechanism essentially requires two basic mechanisms. The first mechanism is one which updates the current earliest deadline, d_j, within each job and ensures that the processor is currently dispatched to the job with the earliest d_j. This is precisely what the UpdateEarliestJobDeadline procedure does in Figure 4.2. The inputs to this procedure are, of course, the job, j, and its new earliest deadline, d_j, a negative d_j representing a nil deadline to indicate that no tasks in j are currently released. After updating the d_j list, this procedure asserts that the currently dispatched job still has the earliest d_j, otherwise it must yield to that job which does. Such a procedure is readily called upon by any task in job j, but there are cases where processes external to j must call upon it, as in the above example in which the system clock handler must release new task instances for j and appropriately update the d_j values for those jobs. These event-driven updates are referred to as *asynchronous* updates, because they might occur when no jobs are active. To fully encapsulate the internal scheduling issues of each job, we assume that each job may have any number of internal event handlers and it must charge the platform with promptly calling upon these internal event handlers when the events occur. Such platform services must be provided by the VaPoR layer in the extended top-layer architecture of Section 3.2, possibly using mechanisms identical to those being implemented today in PERC of Section 2.2.5. The EventsHandler procedure of Figure 4.2 comes to emphasize the need for such a mechanism which must be incorporated into all event-driven services provided by the platform. As illustrated by the simple example of Section 4.1.1, the platform overheads incurred by these services must be taken into account by all internal job scheduling solutions, anyway. The platform event handlers call upon the internal event handlers within the jobs, which must subsequently call upon the UpdateEarliestJobDeadline procedure if these events release new task instances. All elements of internal job scheduling are thus self-contained within the jobs, and all job dispatches are essentially instigated by calling upon the UpdateEarliestJobDeadline procedure. It is via this procedure that low-level constraints are online transmitted to the upper time-sharing layer by the internal job tasks and event handlers.

4.1.3 Time-Sharable Scheduling Policies

It is therefore crucial that all jobs are reliable in calling the procedure on time and with true parameters. It is also crucial that the internal job scheduling policy be bandwidth-bound, i.e. that it never consumes more than the minimal f_j rate of machine cycles even when run on a dedicated processor with a capacity greater than f_j. Clearly this is the case in the above example, as it would be in any deterministic on-demand policy (Section 2.2.1), e.g. RM and EDF, since all tasks in τ_j have already been found to be schedulable by δ_j in a bandwidth of f_j. The same can be said regarding offline planning schemes such as a CE, if UpdateEarliestJobDeadline is called upon before each scheduled activity. On the other hand, a CE implementation which polls the system clock between activities will consume whatever bandwidth it is allocated, so it would not be appropriate for this real-time time-sharing scheme. A bandwidth-bound scheduling policy must essentially relinquish control to the time-sharing layer whenever it has nothing productive to do and whenever it exceeds the allocated rate of a machine cycles. Special care must therefore be applied when using online planning policies (Section 2.2.1) and when using speculative policies (Section 3.3.6). One way of doing this might be to use *bandwidth preservation* techniques [LSS87, SSL89] to ensure that the allocated bandwidths are never over-extended.

4.1.4 Hard EDF Guarantees

Having used a simple example to demonstrate some of the principles, we now prove that EDF interleaving can indeed *hard* guarantee that all internal job schedules will be maintained. Some of these results are trivial, but the notion of real-time time-sharing is new, so these results are brought here for the sake of completeness.

The flow of analysis is as follows. First we assume *continuous* time, i.e. non-atomic machine cycles, job preemptivity at any point in time, no preemption costs, and no interleaving overhead, to show that EDF interleaving maintains the scheduling properties of each job. We then show how job preemption costs, e.g. EDF interleaving overheads and additional preemptivity limitations, are accounted for. Finally we show that augmenting the δ_j scheduling overhead, o_j, by one machine cycle is all we need to accommodate *discrete* time, i.e. atomic machine cycles.

Let us first assume continuous time, job preemptivity at any point in time, and no preemption costs. Let $\mathcal{S}_j$ denote a feasible schedule for all job j tasks, τ_j, when run on a dedicated processor with minimal job capacity, f_j, then $\mathcal{S}_j$ comprises of a sequence of non-overlapping *activity frames.* Each activity frame has a well-defined start time and end time, and also requires an execution time measured in units of atomic machine cycles. If n_i represents the number of machine cycles required by activity i, and f_j denotes the processor capacity assumed by $\mathcal{S}_j$, then the execution time C_i for activity i will be equal to $C_i = n_i/f_j$. As previously described in Section 4.1, each internal δ_j job scheduler must inform the local EDF interleaver of its earliest d_j deadline. To simplify our analysis, we assume here that each δ_j provides the interleaver with *all* of its activity frames and lets the interleaver

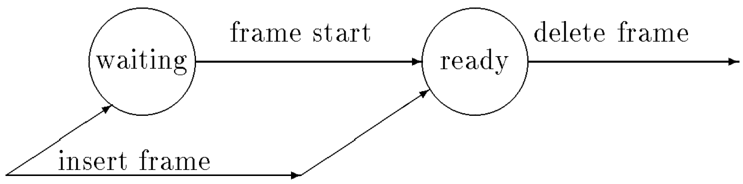

Figure 4.3: Activity Frame Life Cycle

treat each activity end time as a schedule deadline. This is simply another somewhat more costly and less private way to implement EDF interleaving which is more convenient for the following explanation.

As depicted in Figure 4.3, each $\mathcal{S}_j$ activity frame has a typical life cycle. When created by δ_j, activity frames are inserted into one of two possible priority queues, *waiting* or *ready*, depending upon whether the frame start time has already arrived or not. *Waiting* frames must be automatically *transferred* to the *ready* queue when their start times transpire, so all *waiting* frames are naturally ordered by ascending start time values. Frames with equivalent start times are internally sorted by end times so that the earliest deadline frame will always be the first to be transferred. *Ready* frames are deleted by their owner jobs when their tasks complete. The *ready* queue is ordered by ascending end times, so that the top *ready* frame is that with the earliest deadline, and the job owning it is always dispatched. A running job is *preempted* only if the top *ready* frame is no longer his, i.e. if it has just completed a task and deleted the top frame or if a *waiting* frame with an earlier end time has just become ready.

Lemma 1: *Given that each job $j \in J$, has a feasible schedule $\mathcal{S}_j$ for a processor with capacity f_j, then the collection of all activity frames in $\{\mathcal{S}_j\}_{j \in J}$ can be scheduled on a processor with a capacity of $\beta_c = \sum_j f_j$.*

Proof. As previously noted, we still assume continuous time and no job preemption costs, so the combined schedule above, is created by allocating time slices of f_j/β_c to each $\mathcal{S}_j$ within each machine cycle. This will provide each $\mathcal{S}_j$ with an effective clock rate of f_j which was what was assumed when $\mathcal{S}_j$ was found schedulable. Thus *all* application deadlines will be met. $\square$

The obvious problem with the above approach is the prohibitive rate of preemptions, knowing that preemption costs are never zero. We therefore seek a "lazy" interleaving scheme which will greatly reduce the number of preemptions.

Lemma 2: *Given that each job $j \in J$, has a feasible schedule $\mathcal{S}_j$ for a processor with capacity f_j, then the collection of all activity frames in $\{\mathcal{S}_j\}_{j \in J}$ can be scheduled on a processor with a capacity of $\beta_c = \sum_j f_j$ using EDF interleaving.*

Proof. The optimality of EDF [Der74] implies that if the collection of schedules can be scheduled by any interleaving scheme as shown in Lemma 1, then it can be equally scheduled using EDF. □

To accommodate non-negligible job preemption costs, we provide the following lemma.

Lemma 3: *Let p represent the number of overhead machine cycles required by job preemption. If all $\{\mathcal{S}_j\}_{j \in J}$ activity frames include $2p$ surplus machine cycles, then an EDF interleaved schedule is feasible and accounts for preemption costs.*

Proof. When using EDF, a job preemption can only occur when a frame has become ready or completed. Hence, each frame can generate no more than two preemption events, and in both cases the preemption costs are absorbed by the surplus machine cycles within each frame. □

Due to the assumption of continuous time, the interleaved schedule created above might still preempt a frame in the midst of a machine cycle, but machine cycles are atomic. An interesting counter-intuitive example arises when job j has a three periodic single machine cycle activities with equal periods of 1.0 microsecond, equal deadlines of 1/3 microsecond, and release time phases of 0, 1/3 and 2/3 microseconds. Such a job is readily accommodated by a processor with a capacity of $f_j = 3$ MHz, since phases and deadlines correspond nicely to machine cycles, yet it cannot be accommodated by a *faster* processor with a capacity of $f_j = 4$ MHz, because machine cycles are atomic and therefore only half of the machine cycles can be exploited. The following lemma helps us relax the continuous time assumption.

Lemma 4: *If for all $j \in J$, τ_j is schedulable by $\mathcal{S}_j$ with a processor capacity of f_j, when the number of assumed execution cycles, n_i, for each activity frame, i, has been augmented by one machine cycle, then an EDF interleaver can maintain all schedules while preempting only in between machine cycles.*

Proof. Assume we have an interleaving schedule for all the jobs with a surplus machine cycle per frame. Using EDF interleaving, a frame preempts another frame at most once only upon reaching the frame release time. If the release time is not at a machine cycle boundary, then we can use up the surplus machine cycle to delay preemption until the end of the current machine cycle. □

It is clear that job preemption is only one of several factors which contribute to interleaving overheads. To fully account for *all* interleaving overheads, we must also account for the priority queue activities of Figure 4.3, i.e. frame insertions, frame deletions, and the timely transfer of *waiting* frames to the *ready* queue. We distinguish here between two types of interleaving activities, synchronous and asynchronous.

Synchronous Time-Sharing Overheads

Synchronous interleaving activities are those which can be carried out by the job tasks themselves as part of their computational loads, i.e. when they call upon UpdateEarliestJobDeadline. Frame deletion, for example, is carried out by each task as it completes. A periodic task can always insert a frame for its following period as it begins, as well as for other tasks whenever appropriate. Frame insertions for event-driven sporadic tasks must be carried out by the tasks which identify the events. Such synchronized overheads are readily accounted for by appropriately augmenting the number of task computational cycles, n_i, for all tasks, $i \in j$.

Asynchronous Time-Sharing Overheads

Asynchronous interleaving activities include the timely transfer of frames from *waiting* to *ready*, which are usually carried out by time-driven interrupt handlers because they cannot be carried out by the job tasks themselves. As already described in the simple example of Section 4.1.1, such activities can be accounted for by modeling them as independent periodic event handlers, then determining the minimal bandwidth required by these handlers, and, finally, by augmenting the job bandwidth accordingly. An alternate approach would be to use a dedicated co-processor for carrying out the asynchronous interleaving activities.

Each interleaving priority queue is best implemented as an *indexed* heap so that frames can be inserted and deleted with $O(\log N)$ time complexity, where N denotes the worst-case number of frames in the list. Job scheduling activities can be designed to minimize and to upper-bound the number of concurrent job-owned frames, so that a worst-case N can usually be *a priori* determined for every valid partitioning of jobs over processors.

Note that the transfer of a *waiting* frame is urgent only if a job preemption is required, i.e. if the deadline of the top *waiting* frame is earlier than that of the top *ready* frame. Such a transfer is referred to as a *preemptive transfer*, and worst-case preemptive transfers can be treated as *synchronous* activities which can be accommodated by augmenting each job with two job preemption costs, as required by Lemma 4. When several *waiting* frames have equivalent start times, only the first (earliest end time) frame transfer might be preemptive, and all other frame transfer rates may therefore be more relaxed. Such considerations can reduce the minimal bandwidth required to accommodate the asynchronous interleaving overheads for each job.

4.2 A Detailed Example

We now discuss a detailed example to illustrate how job bandwidths are computed and how EDF job time-sharing works. As listed in Table 4.2, this example has three sample jobs, each job having three or four *hard* tasks, e.g. $\tau_1 = \{A_1, B_1, C_1, D_1\}$. As listed in Table 4.3, each job has chosen a different internal scheduling policy: δ_1 for Job 1 is a dynamic RM+PCP (Section 2.2.1), δ_2 for Job 2 is a dynamic EDF (Section 2.1.5), and δ_3 for Job 3 is a static CE (Section 2.1.7). Section 4.2.1 begins by analyzing the worst-case EDF interleaving overheads so that Section 4.2.2 can then demonstrate how job bandwidths are independently computed for each job to accommodate these overheads. Section 4.2.3 the walks through the EDF interleaving process, demonstrating how the job time-sharing mechanisms would function in this example.

4.2.1 EDF Time-Sharing Overheads

As described in Section 4.1.2, to facilitate EDF time-sharing, the time-sharing control layer must maintain a list containing the earliest deadline, d_j, for each job, $j \in J_c$, and each black box job scheduler must ensure that its d_j deadline is up-to-date by calling upon procedures such as those outlined in Figure 4.2.

For efficient maintenance, all of these deadline values are stored in an indexed heap. Letting N denote the maximum number of jobs in J_c, then the

	Job 1				Job 2			Job 3		
task	A_1	B_1	C_1	D_1	A_2	B_2	C_2	A_3	B_3	C_3
period (msecs)	40	60	140	200	80	120	240	120	120	120
deadline (msecs)	40	64	140	120	80	120	240	20	40	60
net machine cycles n_i	30K	46K	46K	6K	33K	33K	33K	18K	36K	54K
max blocking cycles		18K							10K	14K
C_i (ms) @ 4 MHz	8	12	12	4	9	9	9	5	10	15
C_i @ f_j (Table 4.3)	16	24	24	8	40	40	40	20	40	60
B_i (ms) @ 4 MHz		4							3	4
B_i @ f_j (Table 4.3)		8							12	16

Table 4.2: A Sample Job Set

j	job number		1	2	3	1_D
δ_j	job scheduling policy		RM+PCP	EDF	CE	
u_j	job utilization factor		0.97	1.00	1.00	0.60
f_j	min processor	dedicated	1.94	0.85	0.90	0.05
b_j	frequency (MHz)	shared	2.02	0.90	0.97	0.07
α_j	time-sharing overhead ratio		1.04	1.06	1.08	1.33

Table 4.3: Computing Job Bandwidths

earliest deadline can always be found at the top with $O(1)$ complexity, while a deadline can be inserted or deleted with $O(\log N)$ complexity. Thus worst-case costs for deadline updates can be *a priori* determined as a function of N. Note that the logarithmic complexity dependency enables us to be somewhat generous in assuming an upper-bound for N. In our example, $N = 3$, and we have estimated that the WCET for UpdateEarliestJobDeadline, not including a possible job preemption, would be the equivalent of 400 machine cycles, while a job preemption will incur 800 additional machine cycles.

As already noted in Lemma 4, the job preemption overheads of 800 machine cycles must be accommodated by adding surplus machine cycles to each task for two possible job preemptions, i.e. when the task completes or is released. The remaining synchronous and asynchronous UpdateEarliestJobDeadline overheads of 400 machine cycles must be added to the task execution times of all periodic and sporadic tasks which call this procedure.

4.2.2 Computing Job Bandwidths

We now refer again to the three sample jobs of Table 4.2 to demonstrate how job bandwidths are computed and how these time-sharing overheads are accommodated. Each of these jobs has three *mandatory* tasks, with Job 1 having a fourth task, D_1, which is *optional.* We first consider Job 1 without the optional task. For simplicity, all task sets in this example are periodic, but those of Jobs 1 and 3 are not independent. The B_i values in Table 4.2 indicate the worst-case blocking times which each task i might impose upon higher priority tasks due to these inter-task dependencies. As indicated by Table 4.3, the dependencies of Job 1 are resolved by adopting an on-demand RM+PCP policy, while potential conflicts within Job 3 were avoided by an offline-planned static CE. We begin with the simpler case of Job 2 in which all tasks are independent.

The EDF Job Bandwidth

Job 2 uses EDF scheduling, because all tasks in τ_2 are independent. As described in Section 2.1.5, the $\tau_2 = \{A_2, B_2, C_2\}$ task set will always be EDF schedulable if $u(\tau_2) \leq 1$. Using Equation (1.1), the minimal processor capacity, f_j, for *any* EDF-scheduled task set, τ_j, with simple task characteristics is equal to

$$f_j = \sum_{i \in \tau_j} n_i / T_i, \tag{4.2}$$

n_i being the number of execution cycles required by each task $i \in j$. Referring to Table 4.2, given that τ_2 task periods, T_i, are 80, 120, and 240 milliseconds, that each task requires 33 thousand machine cycles, and that each task requires one thousand additional machine cycles to facilitate the internal EDF scheduling, then $f_2 = (33+1)/(1/80+1/120+1/240) = 0.85$ MHz.

As described in Section 4.2.1, to accommodate the time-sharing overheads, we must add 1600 additional machine cycles to each task for two possible job preemptions, as well as 400 additional machine cycles per task for calling the UpdateEarliestJobDeadline when each task completes. Using Equation (4.2) with augmented n_i values, we obtain an augmented bandwidth of 0.89 MHz for this job. As described in the simple example of Section 4.1.1, we also require an additional 25 KHz clock handler for releasing the three tasks at appropriate 40 msec intervals with additional UpdateEarliestJobDeadline overheads, adding an additional bandwidth of 0.01 MHz. Thus, $b_j = 0.89 + 0.01 = 0.90$ MHz.

The CE Job Bandwidth

As previously mentioned, Job 3 activities are not independent, but a static CE schedule resolves these dependencies by breaking up these activities into a fixed periodic sequence of three tasks, $\tau_3 = \{A_3, B_3, C_3\}$. Dispatching these tasks normally incurs a negligible scheduling overhead of only 20 machine cycles per task, so that when running on a dedicated processor, the minimal processor capacity should be equal to the sum of required machine cycles divided by the CE period, i.e. $f_j = (18+36+54)/120 = 0.9$ MHz. Each task in the CE cycle has a deadline determined by the CE schedule, so we must add a UpdateEarliestJobDeadline call at the end of each task, we must accommodate an 50 KHz clock handler to release each task at an appropriate 20 msec interval, and we must accommodate two possible job preemptions throughout any given task execution. Thus, the augmented bandwidth should be equal to $b_2 = f_2 + 3 \cdot 2/120 + 0.4/20 = 0.972$.

The RM Job Bandwidth

We first consider Job 1 without the optional task, D_1. Net task computational requirements for task i are provided in Table 4.2 as numbers of execution cycles, n_i. When testing the schedulability of each job task set, the δ_j schedulers must also consider worst-case overheads for internal job scheduling, e.g. for task preemption, also provided in execution cycles. RM+PCP scheduling overheads are generally lower than EDF scheduling overheads and higher than CE scheduling overheads, so we have augmented the net n_i number of machine cycles per task by 200 additional machine cycles for possible thread context-switching overheads. To evaluate the schedulability of $\tau_1 = \{A_1, B_1, C_1\}$, we employed RMA (Rate Monotonic Analysis) [KRP+93, LSD89], to consider all the relevant RM+PCP factors, e.g. task periods, release times, deadlines, computation times (C_i), maximum blocked times (B_i), and these context switching overheads. RMA analysis is non-trivial, so it has to be conducted for several processor capacities, using an iterative binary search technique to converge on the *minimal* capacity, f_1, which can guarantee τ_1. Such a search is readily automated, but we had to do this manually, whereupon f_1 was approximated as being equal to 1.94 MHz. As in Job 2, the Job 1 bandwidth, b_1, must also accommodate worst-case time-sharing overheads by augmenting all task n_i values by 1600 + 400 machine cycles. Augmenting the n_i values required a new sequence of RMA tests to converge to a new minimal capacity, approximated to be 2.00 MHz. The subsequent bandwidth must then be augmented to accommodate an additional 50 KHz clock handler for releasing the job tasks at appropriate 20 msec intervals with additional UpdateEarliestJobDeadline overheads, adding an additional bandwidth of 0.02 MHz. Adding the additional 0.02 MHz, we obtain $b_1 = 2.02$ MHz.

Going back to the optional D_1 task, we can model D_1 as a separate suspendable Job 1_D, with an independent bandwidth, b_{1_D}, equal to the task $n_i = 6000$ augmented by 2000 time-sharing overhead machine cycles, divided by the 120 millisecond task deadline, i.e. $b_{1_D} = 0.067$ MHz. Jobs 1 and 1_D must then be presented to the load-balancing layer as a mini-cluster with a total bandwidth of 2.09 which must be allocated together to the same processor.

Another alternative way to handle optional Task D_1 would be to model Job 1 as having two modes. The lower bandwidth mode would assume that D_1 is not running and would therefore have a bandwidth of 2.02. The higher bandwidth mode would carry out an RMA analysis on all four tasks, coming up with a bandwidth which might be somewhat less than that of the above mini-cluster. The higher bandwidth mode would also have a higher reward value to reflect the added gain contributed by task D_1. The meta-controller would then have to choose between the two job modes in accordance with available processor capacities.

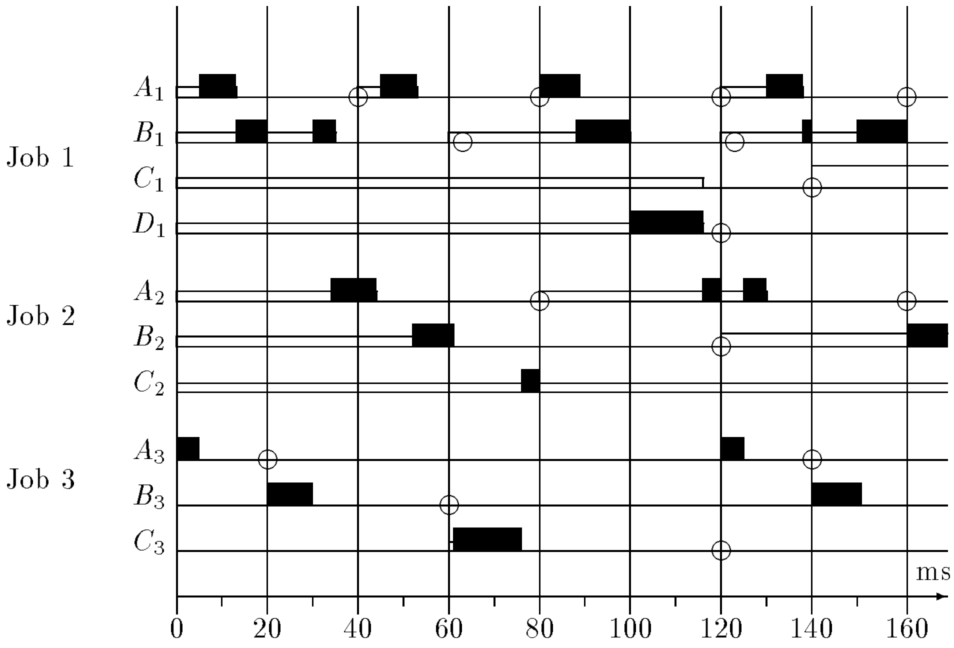

Figure 4.4: EDF Job Time-Sharing

4.2.3 EDF Time-Sharing Walk-Through

Figure 4.4 illustrates how the jobs in this example are EDF dispatched by the time-sharing mechanisms of Section 4.1.2 when sharing a 4 MHz processor, with the subsequent C_i task execution times approximately equal to those of Table 4.2. To the right of each task in Figure 4.4, we have frames which indicate when each task was released and when it completed. Following each frame we find a circle which marks the actual d_j deadline associated with that frame. The shaded areas within these frames indicate when the d_j deadline associated with that frame was the earliest of all d_j deadlines, causing the corresponding job to be dispatched by the EDF interleaver. At time 0, the earliest deadlines for each job are $d_1 = 40, d_2 = 80, d_3 = 20$. Task A_3 is the first to be dispatched because Job 3 has the earliest deadline, but once it completes, d_3 is nil because it has no ready tasks, so Job 1 has the earliest deadline and is therefore dispatched. When Task A_1 completes, $d_1 = 64$, still earlier than $d_2 = 80$, so Job 1 continues and internally schedules Task B_1. Task B_1 is interrupted when Task B_3 becomes ready

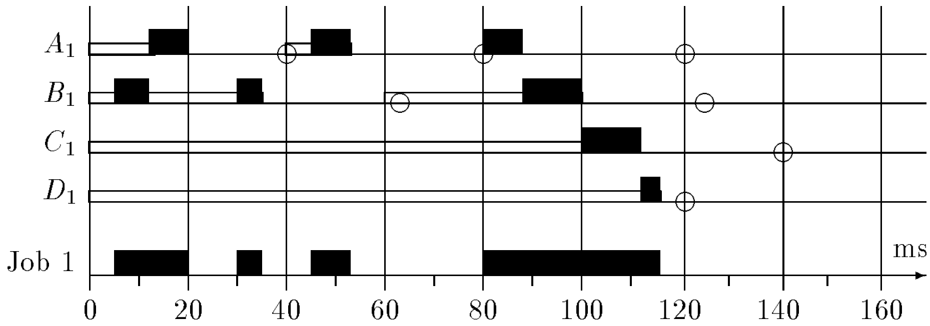

Figure 4.5: Actual RM+PCP Scheduling in Job 1

at 20 msecs, whereupon $d_3 = 60$ which is earlier than $d_1 = 64$, so Job 3 is dispatched. When B_3 completes, B_1 resumes.

Jobs are thus EDF dispatched, but the internal tasks are actually scheduled in accordance with the δ_j scheduling policies employed by each job. To illustrate this point, we provide Figure 4.5 which shows how RM+PDP scheduling is carried out for the *four* tasks of Job 1 whenever Job 1 is dispatched. The shaded areas in this case indicate when each *task* was actually scheduled in accordance with the internal RM+PDP scheduling policy being used. Letting $T(i)$ denote the period for task i, then $T(A_1) < T(B_1)$, so Task A_1 has a higher priority than Task B_1 in an RM policy. Nevertheless, Task A_1 is initially blocked by Task B_1, so Task B_1 initially starts running at time 5 even though Job 1 is then dispatched due to Task A_1's earlier deadline. Task A_1 takes over at time 12 as soon as it gets unblocked, calling UpdateEarliestDeadline when it completes, whereupon Job 3 is dispatched because $d_1 = 64 > d_3 = 60$. Also $T(C_1) < T(D_1)$, so Task C_1 has a higher priority than Task D_1, Task C_1 is therefore scheduled at time 100 msecs, even though it was the d_j associated with Task D_1 which prompted Job 1 to be dispatched at that time due to the earlier Task D_1 deadline. Job schedules are thus *interleaved* without changing or violating the original δ_j task schedules within each job.

This EDF real-time time-sharing scheme might seem trivial at first glance. As already described, however, job bandwidths must be *a priori* computed such that time-sharing overheads be fully accommodated for *arbitrary* J_c job sets, and these time-sharing overheads be minimized. Conventional time-slicing, for example, is found to be completely inadequate, since we can always devise a J_c task set which would require very minute time-slices, thereby incurring unacceptable time-sharing overheads. The advantage of EDF time-sharing is that it upper-bounds the maximum number of job context-switches so that worst-case time-sharing overheads can indeed be absorbed by the job bandwidths regardless of how they are load-balanced over the available platform processors.

4.3 Maximizing Time-Shared Optimality

When contemplating the optimality of real-time time-sharing, several aspects must be considered. First and foremost, one must consider the optimality of the internal job bandwidth utilization. We define the job *bandwidth utilization*, u_j, for job j to be the processor utilization which would be obtained when running job j on a dedicated processor with the minimal capacity, f_j. It follows from Equation (4.1), that the overall processor utilization in the RtTS time-sharing scheme is generally only as good as the u_j bandwidth utilizations provided by each δ_j solution. In the Table 4.2 example, all u_j bandwidth utilizations are near-optimal, so these high processing utilizations are fully maintained when time-shared. Had we obtained poorer bandwidth utilizations for each job, then the overall time-shared utilization would be just as poor. Thus overall time-shared utilization is generally only as good as the *sum of its parts*, and this is an inherent weakness of the time-sharing strategy.

A second issue is the time-sharing overhead factor. The $\alpha_j = b_j/f_j$ values in Table 4.3 provide us with a simple measure for evaluating the overheads introduced by EDF time-sharing. As indicated by the α_j values in this example, the time-sharing overheads can be quite reasonable when the number of machine cycles, n_i, required by the tasks are relatively large, whereupon the additional job time-sharing overheads are only a few percent. For much smaller n_i values, job time-sharing overheads may become prohibitive.

In this section we seek methods to maximize the time-shared utilization divided by the α time-sharing overheads. In Section 4.3.1 we discuss the issue of how to decompose a program into jobs which will maximize the time-shared utilization, and in Section 4.3.2 we show how bandwidth within a job can be maximized. Section 4.3.3 then seeks ways to minimize the time-sharing overheads, while Section 4.3.4 seeks ways to best exploit internal and external bandwidth fragments which are created by job-oriented time-sharing and load-balancing. Implications to solution optimality are

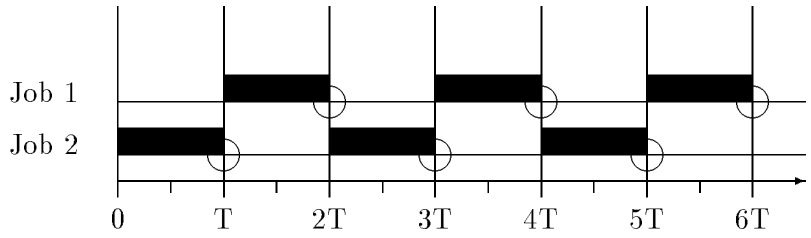

Figure 4.6: Optimality Counter Example

summarized in Section 4.3.5.

4.3.1 Effective Program Decomposition

The detailed example of Section 4.2 demonstrates how an effective system decomposition into jobs can achieve very high bandwidth utilizations. Figure 4.6, on the other hands, illustrates how a poor decomposition can be far from optimal. In this example we have two single-task jobs, with each task having a period of T, an execution time of n cycles, and a deadline of $T/2$. The tasks differ in that the task of Job 1 is always ready at times $t_i = (i + 0.5) \cdot T$, while the second is always ready to run at times $t_i = i \cdot T$. Thus decomposed, each job can only achieve an independent bandwidth of $b_j = 2n/T$, with a bandwidth utilization of only 0.5. Had both tasks been merged into a single job, we would have obtained an optimal bandwidth utilization of 1 by employing a simple CE scheduling policy. A prominent RtTS design goal would be to maximize the bandwidth utilization for each job so that overall time-shared utilization will indeed be *near-optimal.* This design goal is generally realistic because the number of soft tasks is usual much greater that the number of hard tasks, thereby bolstering scheduling flexibility and increasing overall processor utilization.

As previously mentioned, probably the most difficult issue is the lack of an absolute metric for measuring the optimality of a given job decomposition. Referring back to the counter example of Figure 4.6, it would appear that one possible approach would be to develop tools for the spectral analysis of internal bandwidth fragmentation, i.e. compute the frequencies and phases where unused bandwidth is most prevalent within each job. This spectral analysis can then serve to find specific tasks which could improve the bandwidth utilization for each job, and perhaps it could also determine which jobs could be merged to obtain better time-shared utilization. These considerations could then be weighed against other considerations such as implications to job load-balancing and online adaptation. In any event, such a tool should prove useful for measuring potential for improved bandwidth utilization.

4.3.2 Optimal Bandwidths

The choice of a δ_j scheduling policy for job j is arbitrary, but an *optimal*[2] scheduling policy is one which would minimize b_j. Letting Δ denote the entire set of bandwidth-bound scheduling policies as defined in Section 4.1.3, and letting b_δ denote the job bandwidth which would be required for each relevant bandwidth-bound scheduling scheme, $\delta \in \Delta$, then an optimal policy δ_j^* for job j would be

$$\delta_j^* = \arg_{\delta \in \Delta} \min\{b_\delta\}. \tag{4.3}$$

Each δ_j in the above example was found to be *optimal* for each job, having provided the lowest possible b_j bandwidths. Referring to Job 1, CE scheduling overhead would produce a smaller internal scheduling overhead, but the non-harmonic periods would result in a poorer u_1 utilization and a subsequently higher f_1 clock rate to guarantee schedulability. It is also unlikely that a combined EDF and PCP scheme [Bak91] could significantly reduce f_1 by improving u_1, since u_1 is already very close to one and the EDF scheduling overhead is greater than RM. EDF, on the other hand, is the optimal choice for job 2, since u_2 would be considerably lower with RM, and CE is optimal for job 3, since it was able to achieve a u_3 utilization factor of one with minimal internal scheduling overhead. Using the C_i values in Table 4.2, the overall utilization factor for all three jobs sharing a 4 MHz cluster would be 0.973, clearly more than the utilization attainable by non-hybrid scheduling schemes. This unusually high utilization is obtained despite the additional interleaving overhead and an unused bandwidth fragment $(1 - \sum_j b_j)$ of 0.025.

4.3.3 Reducing Time-Sharing Overheads

As already discussed in the beginning of this chapter, and as illustrated by the simple example of Section 4.1.1, there are special cases in which

[2] In a similar manner, the criteria of minimal job bandwidth might also be used to measure the quality of task partitioning into jobs, i.e., an optimal task partitioning into jobs is one which minimizes the sum of all job bandwidths in the application.

job time-sharing can be accomplished without any time-sharing overheads, whatsoever.

The real-time time-sharing scheme is inherently preemptive in nature, so it would seem inappropriate for platforms where multiple tasks or threads are not readily supported. There are special cases, however, where EDF time-sharing can be accomplished without job preemption at all. For the more general case, however, the primary means for reducing time-sharing overheads are in using lightweight threads to minimize job preemption tasks, minimize the system clock handling rate to reduce asynchronous overhead bandwidth, minimize the number and frequency of tasks to reduce the rate of possible job preemptions, and maximize their lengths so that time-sharing overheads are relatively smaller.

4.3.4 Exploiting Bandwidth Fragments

Time-shared utilization is also improved by seeking ways to exploit *external* capacity fragments and *internal* bandwidth fragments, to improve the system gain. External capacity fragmentation is obtained when the load-balancing process does not succeed in allocating the entire processing capacity, β_c, either because the total sum of job bandwidths is less than β_c, or because the remaining job bandwidths do not fit in the remaining processor capacities. Internal bandwidth fragmentation refers to the $1 - u_j$ bandwidth fragments within the jobs which are not utilized by the internal job schedulers.

Both forms of fragmentation can be exploited by ensuring that each job has a sufficient number of soft tasks to run when no hard tasks are scheduled. When considering the gain contributions of each of these soft tasks, time-shared gain might actually be *better* than the sum of its parts, i.e. higher than the overall gain which would have been obtained by running each Job j with a utilization of u_j on a dedicated processor with a processor speed of b_j. When time-sharing on processor c, the internal surplus bandwidths of J_c can be better utilized collectively by using global *best-effort* [Loc86] scheduling techniques to run the higher-reward soft tasks within the *entire* set of jobs, J_c. This requires, of course, that these soft tasks and their rewards be made known to the time-sharing layer by an appropriate mechanism. The time-sharing layer can readily identify these capacities when there are no active d_j deadlines in its deadline heap. Bandwidth preservation techniques [LSS87, SSL89] might even serve optional *hard* tasks in a similar manner.

It is interesting to note that time-sharing decreases total processor utilization in the presence of internal bandwidth fragments, while it can increase the total processor utilization by reducing the external capacity fragments. Thus, for example, a job with two modes, the lesser requiring a bandwidth, b_j, of 8 MHz and the higher being a higher reward mode requiring $b_j = 11$ MHz. If available β_c capacities are 10 MHz and 20 MHz, then we could allocate the 10 MHz capacity to run this job in its lesser mode, or we could allocate the 20 MHz capacity to run the higher mode. In the former case, capacity fragmentation would be 2 MHz, in the latter case it would be 9 MHz. Using time-sharing, we could actually run *two* of these jobs on the 20 MHz processor, one using the lesser mode and the other using the higher mode, with a fragmentation loss of only 1 MHz. Thus real-time time-sharing can increase system gain and and total processing utilization by reducing the capacity fragmentation throughout the system.

4.3.5 Time-Shared Optimality

As described in Section 2.2, optimal scheduling solutions for complex real-time systems are evasive, so we have no absolute metrics for evaluating the optimality of a given solution. Nevertheless, we *do* know that the time-sharing solution of Table 4.2 is nearly optimal if not fully optimal, because the bandwidth utilizations, u_j, are close to one, and the time-sharing overheads are only a few percent. If not for our time-sharing scheme, we would have to apply a uniform scheduling policy, e.g. EDF+PCP, to the union of *all* tasks in the example, which would surely produce a *lower* total utilization, since it would have to consider worst-case blockages such as those which would be introduced by τ_3 without CE.

For lower bandwidth utilizations, we might try to use spectral analysis of internal bandwidth fragments to improve job decomposition, or we might try different internal scheduling policies to reduce the bandwidth. Both of these activities might be very time consuming, but at least they can be carried out offline.

When maximizing system gain, job time-sharing has numerous inherent advantages. Firstly, jobs allow us to apply different scheduling criteria to different subsystems, so that task values can be considered whenever possible. Secondly, it enable us to select the scheduling policy for each job which will maximize its bandwidth utilization. Thirdly, it enables several jobs to share any given processor, and as we have shown, when compared with a single job per processor approach, job time-sharing has inherent potential for increased system gain and reduced capacity fragmentation.

4.4 Conclusions

By definition, *optimal* scheduling solutions can only be obtained when simultaneously considering the characteristics and constraints of *all* tasks in the system. When dealing with complex task characteristics, optimal solutions are rarely feasible. As already described in Section 2.2.1 optimal resolution of complex scheduling problems is generally NP-hard and online overheads for practical approaches are often prohibitive. Practical heuristics are also limited in their inability to guarantee schedulability for dynamic sets of critical tasks. We contend, therefore, that since optimal scheduling is generally irrelevant for complex real-time systems, then there is very little point in an integrated real-time analysis at the task level. There is also very little chance that optimal processor utilizations can be determined for these systems, so we have no absolute metric for the optimality of a practical scheduling solution.

Real-time time-sharing is a new concept in real-time computing and the theoretical analysis presented in this chapter is only a beginning. Time-sharing overheads might be significant for relatively small tasks, but it offers a viable solution, and jobs can often be designed to minimize these overheads. Many of the results presented here have been verified by Monte Carlo simulation using classical RM, EDF and SLF scheduling policies, but we have yet to experiment with a wider range of scheduling policies. Ultimately, the ATLAS testbed should provide us with vital practical experience, but the current initial version does not yet support multi-threading, so we have devised a special non-preemptive time-sharing scheme, described in Chapter 8.

Chapter 5

Load Balancing

Having resolved the issue of job time-sharing, the next issue on our agenda is job *load-balancing*, i.e. how to best allocate a given set of jobs, J, over a given set of shared-memory processors, C. When dealing with varying job states and multiple job modes, job load-balancing is always carried out when the job state is already known and the job mode is already chosen. Thus the load of each job, $j \in J$, is fully characterized by its current bandwidth, b_j, while available resources on each processor, $c \in C$, are fully characterized by the processor capacity, β_c.

Let J_c be the subset of jobs allocated to processor $c \in C$, then each J_c is referred to as a job *cluster*, and all J_c subsets are disjoint by definition. The *primary* load-balancing objective is to seek a a partitioning of J over C such that all jobs on each processor will be schedulable by an appropriate time-sharing scheme. Recalling the time-shared schedulability criteria of Equation (4.1), we seek a set of J_c subsets over C such that

$$\bigcup_{c \in C} J_c = J \tag{5.1}$$

and

$$\forall c \in C : \sum_{j \in J_c} b_j \leq \beta_c. \tag{5.2}$$

A secondary objective, beyond the scope of this thesis, might be to also *balance* the processor utilizations, i.e. to minimize the variance of the $\sum_{j \in J_c} b_j / \beta_c$ utilizations over C. As described in Section 3.1.2, this secondary objective would appear to be superfluous in a real-time context, yet we believe that it can be beneficial for platforms with non-deterministic elements, as we will discuss in Chapter 9. This chapter focuses only on the *primary* load-balancing objective, referred to, here, as job *clustering*. An arbitrary job set J is considered *clusterable* if there exists a partitioning of J over C such that the above equations hold.

The following sections provide solutions for two related problems. Section 5.1 addresses the basic job *clustering* problem, just described, outlining alternatives, presenting Monte Carlo results, and showing how the most cost-effective solution might be chosen for each given problem. Section 5.2 then examines the job *load-shedding* problem which shows how *overload* situations might be dealt with when each job has a single fixed mode, i.e. no lower bandwidth modes are available. Conventional approaches shed tasks on a FCFS (first-come-first-served) basis, and we show how this might effectively be accomplished using a value-based approach at the job level. Section 5.3 concludes with an evaluation of the results and suggestions for future research.

5.1 Job Clustering

We now address the basic job *clustering* problem of how to find a set of disjoint job subsets, $J_{c \in C} \subset J$, which cover the entire set of jobs, J, as in Equation (5.1), while maintaining the time-shared schedulability of all job subsets, as in Equation (5.2). Referring back to the detailed time-sharing example of Section 4.2, a job set with bandwidths of $b_{1+1D} = 2.09$, $b_2 = 0.90$, and $b_3 = 0.97$ is clearly clusterable over a single processor with a capacity of 4 MHz or a pair of processors with capacities of 2.1 MHz each. Note that from a load-balancing perspective, a job *mini-cluster*, e.g. that of Job 1 and Job 1D in the above example, is simply treated as a single job with a bandwidth equal to the sum of job bandwidths within the mini-cluster.

Section 5.1.1 begins by showing that the job clustering problem is a variant of the classic NP-complete binpacking problem, which has several effective approximation algorithms such as FF (first-fit) and BF (best-fit), and their decreasing counterparts, FFD and BFD. The cost-effectiveness of these alternatives are then evaluated in Section 5.1.2. Here, we propose a model for estimating the probability that a given clustering policy will indeed succeed when provided with a given set of job bandwidths and processor speeds. This probability plays an important role in deciding which clustering policy to use and also in choosing which job mode combinations ought to be evaluated by the meta-controller policies of Chapter 6. Implications to *critical* job load-balancing are discussed in Section 5.1.3.

5.1.1 Binpacking Algorithms

In the classical *binpacking* problem [GJ79] we ask a very related question: what is the minimal number of bins (processors), N, required to contain a given number of items (jobs) with arbitrary given sizes (bandwidths) per item? The clustering problem differs only in that processor (bin) capacities need not be uniform and the number of processors is limited. Thus any algorithm which could cluster $\mathsf{njobs} = |J|$ jobs over $\mathsf{ncpus} = |C|$ processors in $O(f(\mathsf{njobs}, \mathsf{ncpus}))$ time complexity, would be able to resolve binpacking in $O(\log \mathsf{njobs} \cdot f(\mathsf{njobs}, \mathsf{ncpus}))$ complexity by carrying out a binary search for N over $[1, \mathsf{njobs}]$. The binpacking problem has already been shown to be strictly NP-hard, so clearly the clustering problem is computationally intractable, as well.

Fortunately, binpacking has several approximation algorithms which have already been shown to be quite effective. It has been shown[GJ79], for example, that FFD (first-fit decreasing) will produce bin numbers which are rarely suboptimal by no more than a few percent, and never exceeds the minimal number by more than 22 percent [JDU$^+$74]. FFD clustering complexity is equal to $O(\mathsf{njobs}[\mathsf{ncpus} + \log \mathsf{njobs}])$ since it entails sorting the jobs by decreasing bandwidth and then allocating the jobs in order, always allocating the job to the first processor which still has enough remaining capacity to accommodate it. It has also been shown that BFD (best-fit-decreasing) produces similar results with an equivalent time complexity, and that mean performance for non-sorting counterparts, FF and BF, is somewhat poorer, with worst-case guarantees of no more than 70 percent more bins than optimal. FF and BF timing complexities are only $O(\mathsf{njobs} \cdot \mathsf{ncpus})$, but the mean execution time for BF is twice that of FF because BF must always scan *all* processors for each job, while FF need only scan *half* of the processors per job on average.

As illustrated by the above discussion, *all* of the conventional binpacking algorithms, BF, FF, BFD, and FFD, can be applied as is to clustering with identical time complexities as binpacking. We note, however, that measures for *suboptimal* performance in binpacking *do not* apply to job clustering, since the first reflects the number of excess bins while the latter reflects the percentage of clusterable job sets which are rejected by the algorithm. To this we devote the following subsection.

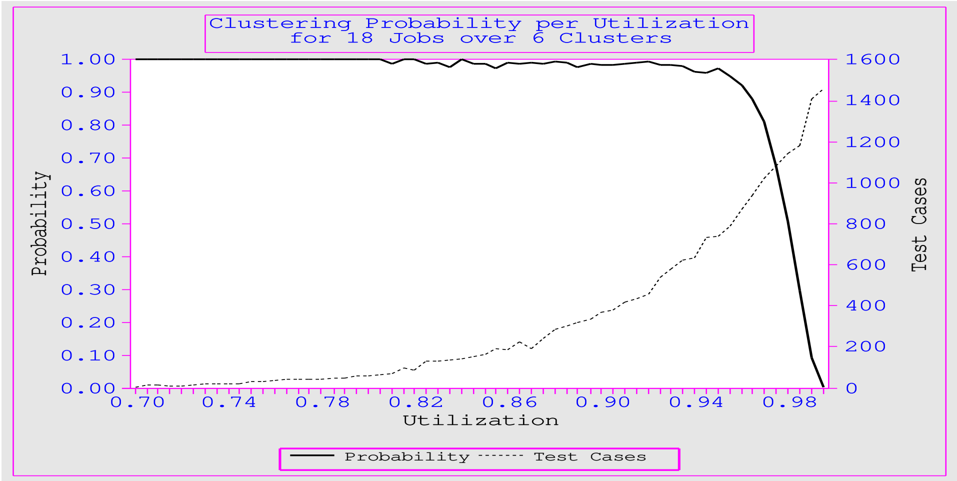

Figure 5.1: Typical FFD Clusterability per Utilization

5.1.2 Estimating Clusterability

As previously noted, performance measures for job clustering are significantly different from those of conventional binpacking. In job clustering, for example, the number of processors (bins), **ncpus** is already fixed, so the necessary total capacity utilization[1],

$$u = \frac{\sum_{j \in J} b_j}{\sum_{c \in C} \beta_c} \tag{5.3}$$

is already known in advance.

Let $\mathsf{clusterable}(\gamma, J, C)$ denote a boolean function which is true when the clustering policy, γ, finds J to be clusterable over C. Then we can define a policy *clusterability*, $\mathsf{clusterability}(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u)$, to be the probability that $\mathsf{clusterable}(\gamma, J, C)$ will be true, given that $|J| = \mathsf{njobs}$, $|C| = \mathsf{ncpus}$, u is the total capacity utilization for J over C as computed by Equation (5.3), and the processor capacities are assumed to be uniform. The clustering *performance* for a given policy, γ, is equal to the probability that γ will

[1] Note that this differs with the overall processor utilization of Section 4.3, in that we are do not consider, here, the internal bandwidth utilization within each job.

verify that J is clusterable over C, given that J is clusterable over C by any *optimal* policy, γ^*, such as by an exhaustive combinatoric search. We can therefore measure policy γ performance, $\Psi(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u)$, for any given u and any given number of jobs and processors, as

$$\Psi(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u) = \frac{\mathsf{clusterability}(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u)}{\mathsf{clusterability}(\gamma^*, \mathsf{njobs}, \mathsf{ncpus}, u)}. \tag{5.4}$$

Letting Γ denote all relevant clustering policies, then we should be able to estimate the performance, $\Psi(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u)$, as well as the computational costs for each policy, $\gamma \in \Gamma$, to determine the policy which would probably be the most cost-effective for each given set of jobs and processors. Such an approach is clearly feasible for reasonably small numbers of jobs and processors, since the clusterability of any given policy, γ, as well as the clusterability of γ^* can be estimated by exhaustive Monte Carlo evaluation. But for larger numbers of jobs and processors, obtaining $\mathsf{clusterability}(\gamma^*, \mathsf{njobs}, \mathsf{ncpus}, u)$ is computationally intractable, since clustering is NP-complete. A possible solution to this problem is developed in the following stages. First we use Monte Carlo analysis to seek a correlation between the clusterabilities of FFD and γ^* when dealing with uniform and non-uniform processor capacities. Note that non-uniform capacities are new to the classical binpacking context. We then seek a correlation between the clusterabilities of any given γ for various ratios of $\mathsf{njobs}/\mathsf{ncpus}$, as well as a relationship between these ratios and clusterability. Finally, we suggest how results might enable us to estimate performance for arbitrary policies and arbitrary sets of jobs and processors.

FFD Clusterability with Uniform Capacities

Figure 5.1 illustrates how the probability that $\mathsf{clusterable}(\mathrm{FFD}, J, C) = \mathsf{true}$, i.e. FFD clusterability, is effected by capacity utilization. The Probability graph is essentially a histogram of Monte Carlo results collected for 200 capacity utilization intervals, when attempting to use FFD clustering to load-balance 10^4 randomly generating bandwidths for $\mathsf{njobs} = 18$ jobs over $\mathsf{ncpus} = 6$ processors with uniform capacities. Using random variable $\mathbf{u}$ to denote these random capacity utilizations, we were not surprised to find that the distribution of utilizations was normal with an average of $\frac{\mathsf{njobs}}{2 \cdot \mathsf{ncpus}}$ and a variance of $\frac{\mathsf{njobs}}{12 \cdot \mathsf{ncpus}^2}$, i.e. $\mathbf{u} \sim N(1.5, 0.041)$. This is represented by the histogram of Test Cases shown in Figure 5.1. As we can see, the probability

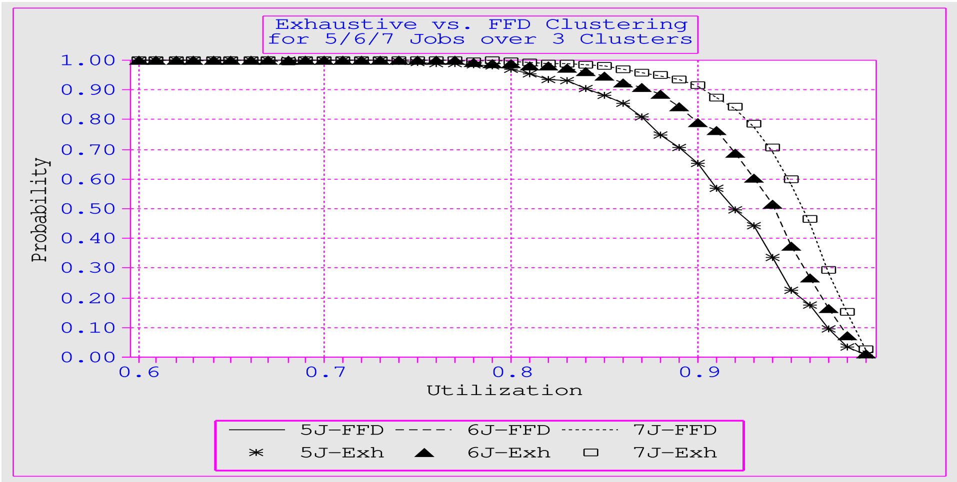

Figure 5.2: Estimating Optimal Clusterability with FFD

of clustering success converges to 1 for utilizations below 0.8, and it goes down to zero as the utilization approaches 1.0.

Estimating Optimal Clusterability with FFD

As previously described, optimal clusterability probabilities are required for absolute measures of clustering policy performance, but the computation of $\mathsf{clusterability}(\gamma^*, \mathsf{njobs}, \mathsf{ncpus}, u)$ is intractable for large numbers of jobs and processors. FFD has already been established as a near-optimal approximation algorithm for binpacking, so we use Monte Carlo analysis to verify whether FFD clusterability might serve as an adequate first-order estimator for optimal clusterability. In our analysis, we could only test for small numbers of jobs and processors, but we did indeed find that for these numbers, the correlation between FFD clusterability and optimal clusterability was virtually 1. It seems reasonable to assume that this is also true for larger numbers of jobs and processors for which optimal clusterability is unattainable.

Some of these results are plotted in Figure 5.2, where the Exh-labeled graphs plot optimal clusterability, obtained by carrying out an exhaustive search over all partitioning possibilities. These histograms summarize

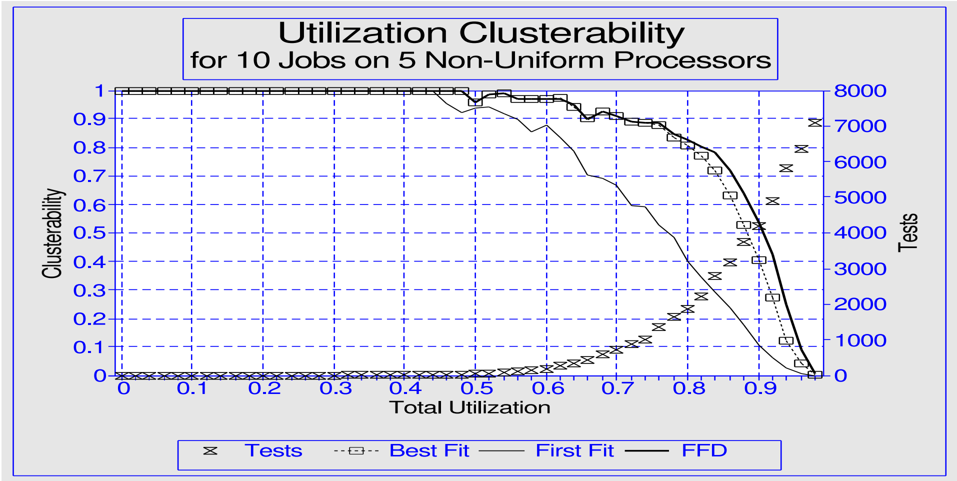

Figure 5.3: Clusterability for Non-Uniform Processors

clusterability results for 100 utilization intervals when attempting to load-balance 10^4 random bandwidth combinations for njobs $= 5$, 6, and 7 jobs over $Ncpus = 3$ with uniform capacities. The number of jobs and processors are predetermined for each graph, and bandwidth values are generated randomly with a uniform distribution over $[0, 1]$.

Though barely noticeable, there are isolated u intervals where FFD clusterability is lower by nearly 2 percent, but they are too small and too few to significantly effect the overall correlation. We suggest that this is ample justification for using FFD clusterability as a first-order estimator for optimal clusterability when processor capacities are uniform. Note that BFD and FFD have equivalent clusterabilities for uniform processor capacities, so we prefer FFD which is somewhat quicker.

Clusterability for Non-Uniform Capacities

The Monte Carlo experiment was then repeated for non-uniform processor capacities, with processor capacities being generated randomly with a uniform distribution over $[0.5, 1.0]$. As we can see in Figure 5.3, policy clusterability differences are now much more pronounced. FF performance is greatly diminished and BF has a significant edge over FF for $u > 0.44$. The

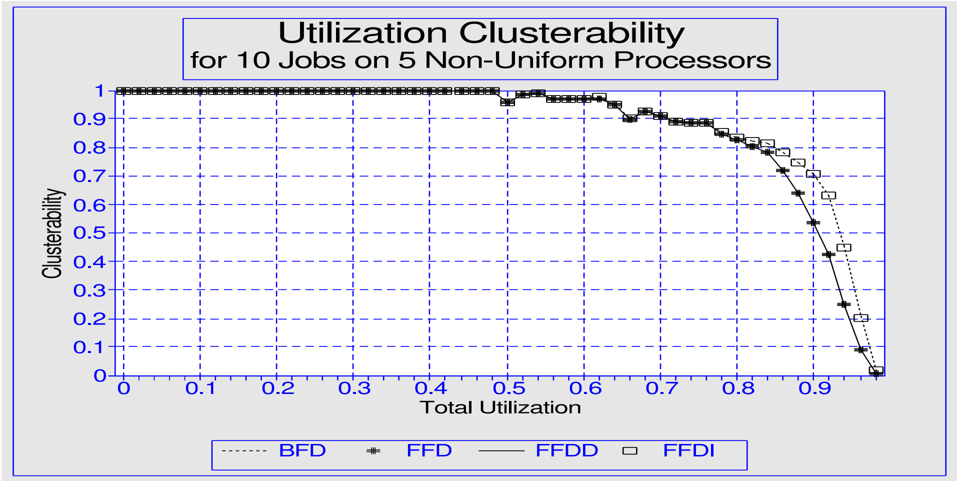

Figure 5.4: Clusterability for Ordered Capacities

u range where the FFD advantage is realized is similar to that of Figure 5.7.

Thus FFD is *not* a good estimator for optimal clusterability when capacities are *not* uniform. BFD, on the other hand, would appear to be a more worthy candidate for the first-order estimation of optimal clusterability when clustering for arbitrary processor capacity distributions.

Pre-Sorted Processor Capacities

Seeing that BF does better than FF with non-uniform capacities, we then asked ourselves how BFD would compare with FFD, and what would happen if we sorted the processor capacities in either increasing or decreasing order before job allocation. The results of this experiment appear in Figure 5.4, where FFDD is an FFD variant with capacities in decreasing order, and FFDI is an FFD variant with capacities in increasing order. Clearly FFDD and FFDI time complexities are greater than that of BFD and FFD complexities by $O(\log \mathsf{ncpus})$. As expected, BFD does somewhat better than FFD. A more curious outcome was that BFD and FFDI had similar performance, as did FFD and FFDD. Thus FFD performance is somewhat enhanced for certain ranges of u, by pre-sorting the processor capacities in increasing order.

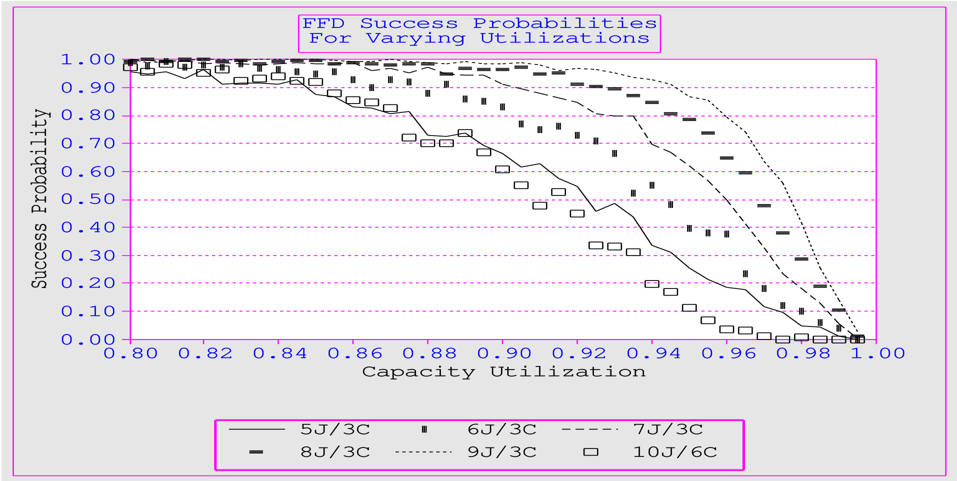

Figure 5.5: FFD Clusterability

We also see that FFD is *not* a good estimator for optimal clusterability when capacities are not uniform. BFD, on the other hand, would appear to be a more worthy candidate for the first-order estimation of optimal clusterability when clustering for arbitrary processor capacity distributions.

FFD Clusterability for Larger Numbers

Additional $\mathsf{clusterability}(FFD, \mathsf{njobs}, \mathsf{ncpus}, u)$ plots from the same experiment are provided in Figure 5.5, illustrating FFD performance for larger numbers of jobs and processors. The 10J/6C plot, for example, is for $\mathsf{njobs} = 10$ and $\mathsf{ncpus} = 6$. As expected, the clusterability (Success Probability) converges to zero as u rises to 1.0, and it converges to 1 as u descends to 0.8. We can see that the ratio of jobs to processors, $\mathsf{njobs}/\mathsf{ncpus}$, has a very significant effect upon the clusterability, with the probability of success increasing with the ratio. These findings clearly indicate that capacity fragmentation losses are bounded, and that they become more and more negligible as the ratio of jobs to processors rises.

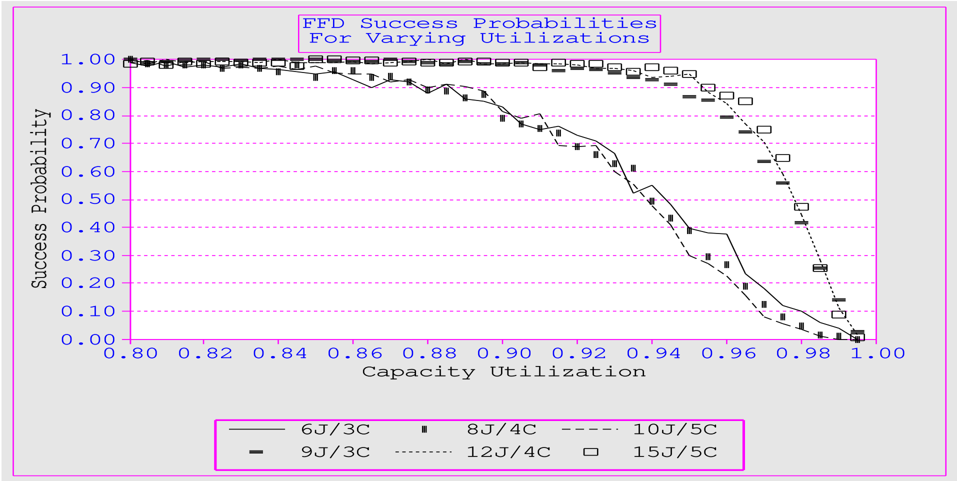

Figure 5.6: Clusterability Dependency on njobs/ncpus Ratios

Exterpolated Clusterability for Larger Numbers

Figure 5.6 shows how job and processor sets with similar jobs-to-processors ratios have similar clusterabilities. Here we see that all plots with ratios of 2 and all plots with ratios of 3 are strongly correlated, so we can expect similar results when the number of jobs and processors are greatly increased. This should enable us to estimating FFD clusterability for arbitrary numbers of jobs and processors, by extrapolating from smaller numbers with similar ratios, and then interpolating.

We also suggest that more extensive research should yield higher-order bias-correction methods which would provide us with more accurate polynomial-time estimates for $\mathsf{clusterability}(\gamma^*, \mathsf{njobs}, \mathsf{ncpus}, u)$. These probabilities could then be derived empirically for large numbers of utilizations and job/processor ratios, whereupon they can serve any given application by interpolating between the probabilities of known utilizations and ratios.

Cost-Effective Clustering

For a comparative analysis of policy performance, policy clusterability is sufficient. The graphs in Figure 5.7 illustrate how $\mathsf{clusterability}(\gamma, 10, 5, u)$

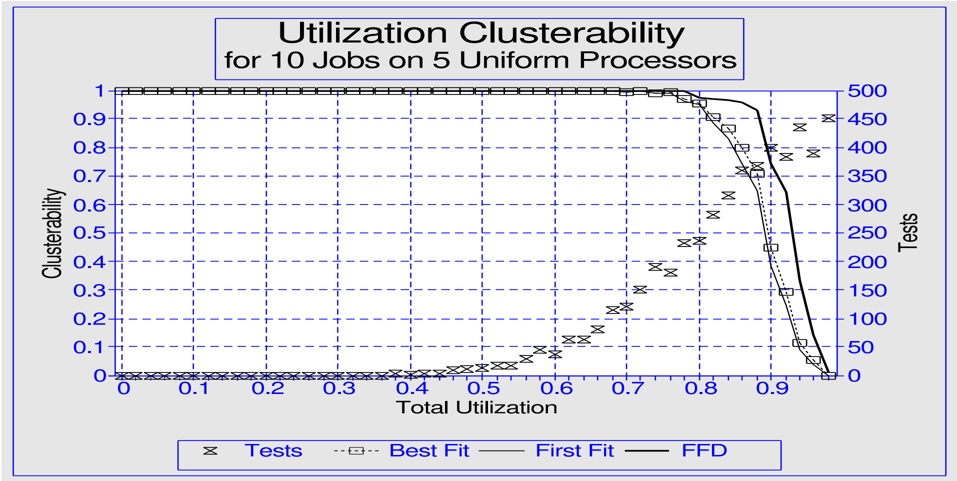

Figure 5.7: Policy Clusterability for 10 Jobs over 5 Uniform Processors

behaves for policies (γ) FF, BF and FFD. Each graph, here, is a histogram of results collected for 50 u intervals, when attempting to load balance $5 \cdot 10^3$ randomly-generated sets of 10 job bandwidths over 5 processors with uniform 1.0 capacities. Job bandwidth values are generated randomly with a uniform distribution over $[0, 1]$. The Tests graph indicates the number of sets which were tested for each u interval. As indicated, policy clusterability for all three policies are very similar for $u \leq 0.8$, and they are similar for BF and FF over the entire utilization range, even though BF has a consistent slight edge over FF. FFD performance is significantly higher that that of BF and FF for $0.84 \leq u \leq 0.96$, but for other values of u we might prefer using the less expensive FF policy.

The same experiment was repeated for clusterability$(\gamma, 12, 3, u)$, i.e. for 12 jobs over 3 processors, and the results displayed in Figure 5.8. Here we see that the higher njobs/ncpus ratio has raised the clusterability of all policies, but the relative performance remains the same. FFD advantages are now primarily realized for a narrower and somewhat higher range of $0.92 \leq u \leq 0.99$.

We can therefore conclude that job load-balancing, or *clustering*, can be carried out using any of the above policies, but that the most cost-effective

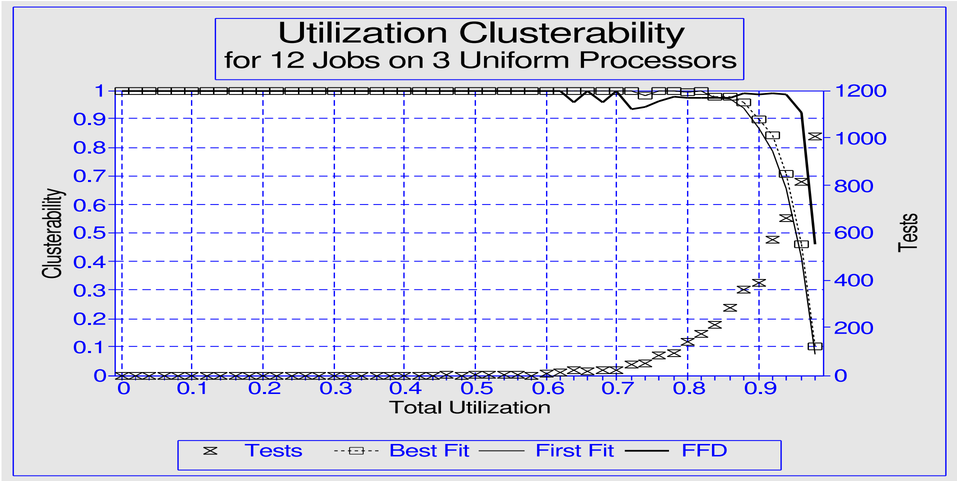

Figure 5.8: Policy Clusterability for 12 Jobs over 3 Uniform Processors

policy is very much dependent on u, the ratio of jobs to processors, and the distribution of processor capacities.

5.1.3 Critical Clustering

When dealing with dynamic *critical* jobs, i.e. jobs with mandatory hard tasks, platform capacities must provide u values which guarantee $\mathsf{clusterability}(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u) = 1$, for the clustering policy γ, being used. Let $\bar{u}$ denote the minimal utilization which guarantees clusterability for a given policy in a given application, i.e.

$$\bar{u} = \min\{u|\mathsf{clusterability}(\gamma, \mathsf{njobs}, \mathsf{ncpus}, u \leq \bar{u}) = 1\}. \tag{5.5}$$

It is readily shown that $\bar{u} \geq 0.5$, and that it grows with $\min\{\beta_c\}/\max\{b_j\}$. As platform capacities grow, this ratio will only get better, thereby increasing $\bar{u}$ with time. Non-critical jobs will only be allocated after the critical jobs have already been clustered, so all we have to ensure is that the sum of minimal bandwidths for critical jobs does not exceed $\bar{u}$, and that job bandwidths do not exceed the individual processor capacities, i.e. $\forall j : b_j \leq \beta_c$. Soft workloads generally outweigh hard critical workloads, so the above constraints will rarely pose a problem.

5.2 Value-Based Load-Shedding

What happens when $\mathsf{clusterable}(\gamma, J, C)$ is false for the current system jobs and processors, and the current clustering policy, γ? Which incoming jobs in J do we reject? Which do we suspend? As previously described in Section 2.2.2, conventional dynamic scheduling approaches will always shed the incoming tasks, so that shedding is essentially done on a FCFS basis and no accepted tasks ever get suspended. A nice feature of the RtTS architecture is that load shedding can be *value-based*, i.e. lower priority jobs may be suspended to accommodate incoming jobs with higher priority. The *load-shedding* problem would then be to determine which jobs should run to maximize system gain. Here we must also consider the job reward, r_j, for each job $j \in J$, in seeking a job subset, $\hat{J} \subseteq J$, such that $\mathsf{clusterable}(\gamma, \hat{J}, C)$ provides the highest system value, assuming that $v = \sum_{j \in \hat{J}} r_j$.

First we show in Section 5.2.1 that job load-shedding on a single processor is equivalent to the classical zero-one *knapsacking* problem. Non-integer zero-one knapsacking is known to be NP-hard but the inexpensive DG (density greedy) heuristic is known to be quite effective [GJ79], so Section 5.2.2 shows how DG can be applied to job load-shedding on a uniprocessor. When load-shedding on a multi-processor platform, we have a composite binpacking and zero-one knapsacking problem, so Section 5.2.3 shows how DG can be extended to accomplish this. The discussion of more sophisticated and expensive polynomial-time schemes is deferred to Chapter 6, where load-shedding decisions are further complicated by the need to simultaneously make job mode selection decisions, whereupon DG-based algorithms are relatively ineffective.

5.2.1 The Knapsacking Problem

Load-shedding on a single given processor, c, is equivalent to the classical *zero-one knapsack* problem, where we are given a positive knapsack capacity β_c and two finite sets of positive integers: $B = \{b_1, b_2, \cdots, b_N\}$ and $R = \{r_1, r_2, \cdots, r_N\}$. The b_is may be interpreted as weights (bandwidths) of the items (jobs) $1, 2, \cdots, N$ and the r_is give the profits (rewards) associated with each item (job): i.e. if item i is put into the knapsack then the profit r_i is earned and the knapsack capacity is subsequently reduced by b_i. The problem is to maximize $v = \sum_i r_i \kappa_i$ subject to $\sum_i b_i \kappa_i \leq \beta_c$ and $\kappa_i = \{0, 1\}$, hence the *zero-one* characterization. In a *multiple-choice* knapsack variant, the N items are further partitioned into several equivalence classes and it is stipulated that no more than one item may be chosen from each equivalence class. In Chapter 6 we discuss the job mode selection problem which is clearly equivalent to this multiple-choice variant when carried out for a single given processor c, since N can represent the total number of relevant modes in J_c and each $j \in J_c$ is an equivalence class from which only one mode can be selected.

It is generally assumed in knapsacking that b_i and r_i are positive *integers*, which is why it is often referred to as *integer* knapsacking. The *integer knapsack* problem has a polynomial-time dynamic-programming solution. Unfortunately this solution cannot be applied, *as is*, to load-shedding because job bandwidths are not represented by integers. and the number of dynamic programming states would grow exponentially. Thus non-integer zero-one knapsacking and multiple-choice knapsacking are both known to be NP-hard.

5.2.2 Density-Greedy Load-Shedding

Fortunately, the zero-one knapsack problem has been extensively studied and several fast approximation algorithms have been devised. A very simple approximation algorithm is the *density-greedy* (DG) algorithm, which has complexity $O(N \log N)$ for N items (jobs), but worst-case returns can be as dismal as half optimal [GJ79].

We first apply DG to load-shedding on a single processor. Like FFD, DG has a preliminary sorting phase where all jobs are sorted by a *decreasing* job density, where the *density* for job j is equal to the job reward divided by the job bandwidth, r_j/b_j. It then enters the greedy phase, where each job is treated in the density-sorted order, and we accept or reject each job, $j \in J$, depending upon whether the remaining capacity in the single processor is sufficient to accommodate the job bandwidth, b_j. Letting $\hat{J}$ represent the growing set of already accepted jobs, then an additional job j is accepted if

$$\sum_{j' \in \hat{J}} b_{j'} + b_j \leq \beta_c. \tag{5.6}$$

The timing complexity for sorting by density is $O(\mathsf{njobs} \cdot \log \mathsf{njobs})$, and the timing complexity of the greedy phase is is only $O(\mathsf{njobs})$, since the current sum of $\hat{J}$ bandwidths in Equation (5.6) is accumulated on the fly. Thus the total DG time complexity is $O(\mathsf{njobs} \cdot \log \mathsf{njobs})$.

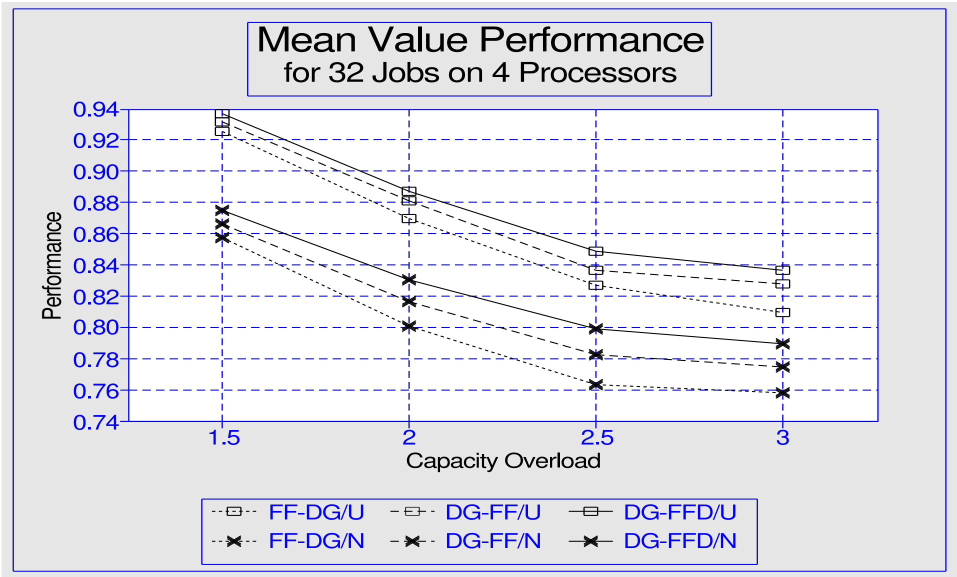

Figure 5.9: Mean Value-Based Load-Shedding Performance

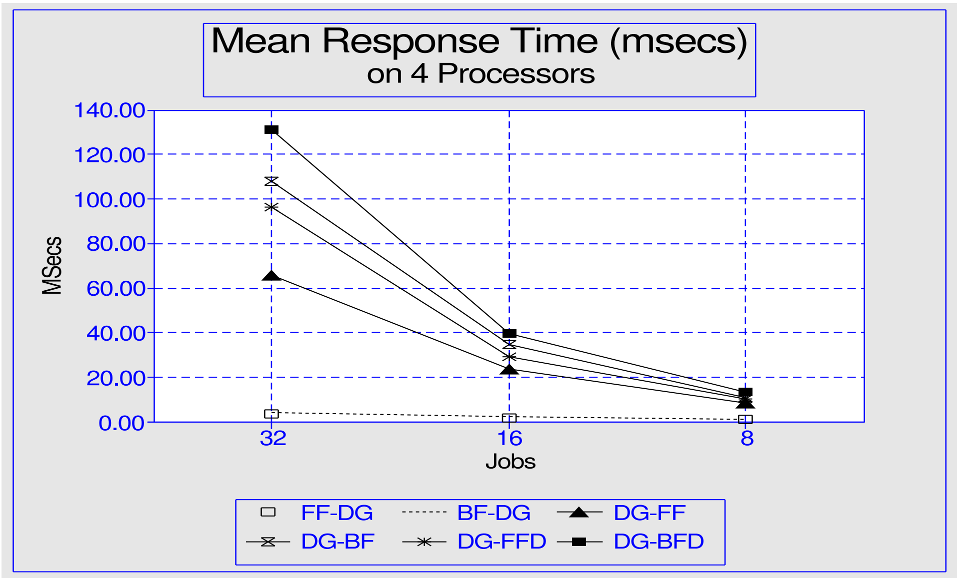

Figure 5.10: Mean Value-Based Load-Shedding Response Times

5.2.3 Combined Load-Balancing and Load-Shedding

The DG greedy phase must be somewhat modified when being applied to a multiple-processor platform since zero-one knapsacking and binpacking must now be carried out simultaneously. One way of doing this is to accept each j in turn only if the set of jobs $\hat{J} \cup \{j\}$ is found *clusterable* over C by a given clustering policy, γ. This DG variant would have a total time complexity equal to njobs times the time complexity of γ. A less costly variant would be to apply simple FF or BF to the pre-sorted density-ordered set of jobs, whereupon the time complexity is reduced to $O(\mathsf{njobs} \cdot [\mathsf{ncpus} + \log \mathsf{njobs}])$.

Figure 5.9 plots mean value performance for three DG variants: FF-DG, DG-FF and DG-FFD, as a function of the capacity overload. FF-DG essentially carries out FF on jobs which have been pre-ordered by decreasing density. DG-FF and DG-FFD are extensions of DG which use FF or FFD, respectively, throughout the greedy phase, to accept or reject each job in turn. Mean value performance was computed by Monte Carlo analysis for both *uniform* and *non-uniform* processor capacities. Graphs depicting mean performance for uniform capacities are labeled with a /U extension, while graphs for non-uniform capacities are labeled with a /N extension. Each graph shows results for 50 randomly generated sets of bandwidths, rewards, and capacities for 32 jobs and 4 processors, whereupon the processor capacities were scaled down to obtain each of the four levels of capacity overload. Bandwidths, rewards, and capacities were all randomly generated with uniform distributions over $[0, 1]$. As illustrated by Figure 5.9, mean value performance decreases as the capacity overload increases, but performance eventually stabilizes. We also find that performance for uniform capacities is better than when load-shedding over non-uniform capacities, that mean DG-FFD performance is only a few percent better than DG-FF, and that FF-DG performance is only a few percent better than DG-FF. Similar graphs were also obtained for the BF (best-fit) counterparts of each variant, but they are not shown in Figure 5.9 because mean performance differences were only fractions of a percent. Figure 5.10, on the other hand, plots the mean response times, for all of these variants, as a function of njobs when ncpus is fixed at 4 processors. All response times were obtained for a 486/33MHz PC. Here we find that the response time behavior for each of the DG variants is consistent with its time complexity. For 32 jobs, FF-DG and BF-DG response times are only a fraction that of the other variants, making it significantly more cost-effective than the other DG variants when catering to large numbers of jobs. We can also see that only a few milliseconds are needed to carry out load-balancing and value-based load-shedding for considerable numbers of jobs and processors.

5.3 Conclusions

In this chapter we have presented several alternate binpacking policies, and variants thereof, for clustering dynamic sets of jobs on a multiprocessor platform. Using Monte Carlo analysis, we have found that policy performance can generally be estimated as a function of the jobs-to-processors ratios, and the total capacity utilization, u. The policy-maker in Figure 3.2 can then consider the policy response times and their expected performance to select the most effective job clustering policy for each system state.

For overloaded system states, we have also presented several alternate DG-based policies for simultaneous job clustering and value-based load-shedding. For adaptive jobs with multiple modes of operation, software meta-control must also determine the job modes which maximize system gain for each system state. To address adaptive systems, Chapter 6 introduces two new near-optimal families of meta-control algorithms, G^2 and QDP, which can simultaneously carry out mode-selection, clustering, and load-shedding decisions. Policy-making issues are then addressed in Chapter 7.

When compared with the conventional *task* load-balancing schemes of Section 2.2.2, *job* load-balancing has several significant advantages. Any of several simple binpacking policies can serve to allocate jobs to processors without recalculating schedules. In an overloaded situation, any of several DG-based policies can be used to shed tasks in a non-FCFS basis, thereby providing higher system *gain* and also obtaining *more predictable* system behavior. The solution is also highly portable and fault-tolerant, enabling the application to automatically adapt to reduced or enhanced system capacities.

When dealing with *critical* jobs, i.e. jobs with mandatory hard tasks, deterministic clusterability guarantees are provided as long as their total bandwidths do not exceed a known platform utilization which is usually better than half. Remaining capacities can then serve non-mandatory jobs in a manner which maximizes system gain. We know of no other load-balancing scheme which can provide deterministic guarantees for dynamic hard tasks when task characteristics are complex.

Future Research

As is, this chapter merely sets the foundations for job clustering and value-based load-shedding, leaving many issues open for future research. In Chapter 7, for example, we develop an analytic model for estimating optimal system values and for predicting QDP performance for arbitrary given system states. Similar analytic models should be devised for the clustering policies presented in this chapter, relating to both uniform and non-uniform processor capacities. For critical job clustering, we can always assume a worst-case $\bar{u}$ of 0.5, but we should be able to assume much higher $\bar{u}$ values when the maximum job bandwidth is only a fraction of the minimal processor capacity.

Also to be considered are alternatives for further reducing the complexities and improving the performance of policies presented in this chapter. In the DG-FFD policy, for example, we assumed that FFD clustering had to be carried out with each attempt to add an additional job to the current set of accepted jobs, $\hat{J}$. Each FFD clustering requires that all jobs be sorted by decreasing bandwidths, but in each subsequent FFD clustering there is only one additional job being considered, so the sorting complexity should be reduced. Another interesting possibility would be to forego clustering of $\hat{J}$ over C as long as clusterability is nearly guaranteed, since adding an additional job to $\hat{J}$ may actually end up being clusterable even if $\hat{J}$ isn't. Research on all of these issues has only begun.

Chapter 6

Meta-Control

Having catered to the meta-controller needs of *single*-mode jobs in Chapter 5, we now address the more complex needs of *adaptive* real-time systems which have a significant number of multiple-mode jobs and an appropriate system gain function, so that it might best accommodate a wide range of program and platform states. Given a specific system state, s, the best-effort meta-controller must select a job mode, $m_j \in M_j(s)$, for each multiple-mode job, $j \in J$, in a manner which maximizes the subsequent system gain, $v = g(\{r_j(m_j, s)\})$, while ensuring that the subsequent job bandwidths, $\{b_j(m_j, s)\}$, are still clusterable over the available processor capacities, $\{\beta_c\}$.

To address the formidable time complexities of making even a single meta-controller decision in such adaptive systems, this chapter introduced two new families of algorithms, QDP and G^2, capable of a near-optimal resolution for each decision. Many important aspects of RtTS software meta-control have already been addressed by Chapter 5, in which we showed how appropriate conventional binpacking and knapsacking algorithms can effectively accommodate simultaneous job clustering and value-based load-shedding for sets of *non-adaptive* (single-mode) jobs. These results now serve as fundamental building blocks for the more elaborate algorithms presented in this chapter. Job clustering must be carried out for each of several job mode combinations being evaluated, and job *load-shedding* is carried out by choosing lower bandwidth modes, including a possible zero-bandwidth *suspension* mode for those jobs which can be suspended.

As already discussed in Section 3.1.2, the notion of *best-effort* is very system dependent. In several of the examples of Section 3.3, we suggested that what we would want is a uniform contribution for all jobs. This could be obtained by optimizing for a product gain function, or by a much simpler heuristic, since when seeking a *uniform* set of job rewards, the number of mode combinations which need to be evaluated is much less. With the ATLAS experiment in mind, this chapter assumes an *additive* system gain function.

In Section 6.1 we provide a formal definition of the adaptive meta-control problem, finding it to be equivalent to a composite binpacking and multiple-choice knapsacking problem. Section 6.2 introduces the QDP approximation algorithm which applies dynamic programming to mode bandwidths which have been quantized by a controllable quantum value, q. Section 6.3 introduces the G^2 approximation algorithm which uses an iterative gradient-greedy approach. Each algorithm is described in detail and time/space complexities are subsequently derived. Both algorithms are shown to have several variations, so they essentially represent two *families* of available meta-control policies. In Section 6.5, for example, we model the second phase of the QDP algorithm as a Markov decision process, whereupon we devise a QDP variation which can optimize performance for bound response times. Nice features of both families is that they have reasonable time complexities, that they provide us with the means to evaluate the performance of their decisions, and that they enable us to trade off time for performance in a predictable manner.

A Monte Carlo evaluation of their performance is provided by Section 6.4, where results for both algorithms indicate that response times can be very reasonable, that typical gain performance ranges of 0.95-0.99 are readily achieved, and that overall behavior is very consistent. We also find that G^2 converges very quickly and that the QDP quantum, q, appears to have a sub-linear affect on performance, i.e. that starting from certain q values, finer q do not contribute to higher system values. We also find that G^2 performance is only slightly lower than that of QDP and at considerably lower cost, and that both perform significantly better than a conventional DG approach in all of our experiments. The two families differ also in the nature of system gain functions which they can accommodate. Comparisons with previous algorithmic efforts are provided by Section 6.6 Conclusions and suggestions for future research are presented in Section 6.7

6.1 The Meta-Control Problem

As described in Section 3.1.2, a meta-control decision is required for each significant transition in the current system state, $s = \{\mathcal{J}, \mathcal{C}\}$, where $\mathcal{J}$ is used to denote the current program state, and $\mathcal{C}$ is used to denote the current platform state. From an RtTS software control perspective, $\mathcal{C}$ is fully specified by the current processor set, C, and by the β_c capacity for each processor $c \in C$. In a similar manner, $\mathcal{J}$ is fully specified by the current job set, J, by the current mode set, M_j, for each job $j \in J$, and by the current reward and bandwidth functions, $r_j(m)$ and $b_j(m)$, which provide the reward and bandwidth for each possible mode $m \in M_j$. For simplicity, all rewards and bandwidth in this chapter will be normalized so that the maximum β_c capacity in the platform is always equal to one.

6.1.1 Mode Vectors

As described in Section 3.1.2, the job mode abstraction is used to introduce several alternatives for accomplishing a given job, whereupon the meta-controller must select an appropriate job mode $m_j \in M_j$ for each job $j \in J$. When carried out for *all* jobs in J, any mode selection decision can be represented by a *mode vector*, ω, of $\mathsf{njobs} = |J|$ integers, where the jth integer, $\omega[j]$, identifies the m_j mode chosen for job j. Assuming an additive system gain function, each such mode vector, ω, has a well-defined system gain, $v(\omega)$, which is computed by

$$v(\omega) = \sum_{j \in J} r_j(\omega[j]). \tag{6.1}$$

In a similar fashion, each ω also has a well-defined total bandwidth, $b(\omega)$, which is computed by

$$b(\omega) = \sum_{j \in J} b_j(\omega[j]), \tag{6.2}$$

whereupon the total capacity utilization, $u(\omega)$, for ω would be equal to

$$u(\omega) = \frac{b(\omega)}{\sum_{c \in C} \beta_c}. \tag{6.3}$$

Letting Ω denote the set of all possible mode vectors over $\mathcal{J}$, then a minimal bandwidth mode vector, $\omega_0 \in \Omega$, is defined by

$$\omega_0 = \arg_\omega \min_{\omega \in \Omega} b(\omega). \tag{6.4}$$

As in Section 5.1.2, we define the boolean $\mathsf{clusterable}(\gamma, \omega)$ function to be true if the set of job bandwidths dictated by ω is *clusterable* over $\mathcal{C}$ by the clustering policy, γ. Let $\Omega_{\mathbf{c}} \subset \Omega$ denote that set of mode vectors which is clusterable by *any* clustering policy, γ, regardless of γ time complexity, i.e.

$$\Omega_{\mathbf{c}} = \{\omega \in \Omega \mid \exists\gamma : \mathsf{clusterable}(\gamma, \omega) = \mathsf{true}\}. \tag{6.5}$$

We can then define $\mathbf{c}(\omega)$ to be an indicator whether $\omega \in \Omega_{\mathbf{c}}$, i.e.

$$\mathbf{c}(\omega) = \begin{cases} 1 & \omega \in \Omega_{\mathbf{c}} \\ 0 & \omega \notin \Omega_{\mathbf{c}} \end{cases} \tag{6.6}$$

6.1.2 Cluster Vectors

As described in Chapter 5, the primary output for any clustering policy is the allocation of each job $j \in J$ to a specific processor $c_j \in C$. The set of jobs allocated to a given processor c is then referred to as a job cluster, J_c. Such a clustering decision is readily represented by a *cluster vector*, $\mathbf{c}$, of $|J|$ integers, where the jth integer, $\mathbf{c}[j]$, identifies the J_c that job j belongs to.

6.1.3 Optimality Criteria

As previously described, a software meta-controller may alter operational modes, shed non-critical loads, and migrate tasks to best accommodate varying runtime circumstances. Translated into an RtTS context, the meta-controller must select job modes and load-balance the subsequent job bandwidths in a manner which will maximize the system gain without overloading any of the processors in the system. Load-shedding is accomplished by equipping each non-critical job with a zero-reward, zero-bandwidth mode, also called a *suspension* mode, whereupon that job is suspended if the suspension mode is selected.

Let $v^*(\Omega, b)$ denote the optimal (maximum) system gain which can be obtained by any mode vector $\omega \in \Omega$, which has a total bandwidth of b or less, i.e.

$$v^*(\Omega, b) = \max_{\omega \in \Omega}\{v(\omega) \mid b(\omega) \leq b\}. \tag{6.7}$$

$v^*(\Omega, b)$ is clearly monotonic over b, i.e.

$$\forall b_1, b_2 : b_1 > b_2 \Leftrightarrow v^*(\Omega, b_1) \geq v^*(\Omega, b_2). \tag{6.8}$$

In a single processor system with a total capacity β, an optimal mode vector, ω^*, would be such that $v(\omega^*) = v^*(\Omega, \beta)$. In a multiprocessor system, we must further verify that any candidate mode vector ω is also *clusterable* over $\mathcal{C}$, i.e. that $\omega \in \Omega_{\mathbf{C}}$. Consequently, for the general multiprocessor case the optimal system gain, v^*, is equal to

$$v^* = \max_{\omega \in \Omega} \{ v(\omega) \mid \mathbf{c}(\omega) = 1 \}. \tag{6.9}$$

whereupon the following holds for an optimal mode vector, ω^*:

$$\forall \omega \in \Omega : v(\omega) \le v(\omega^*) \text{ or } \mathbf{c}(\omega) = 0. \tag{6.10}$$

6.1.4 Computational Intractability

The cardinality of Ω is $\prod_{j=1}^{|J|} |M_j|$, so an exhaustive search for an optimal mode vector, $\omega^* \in \Omega$, is computationally intractable. The cardinality of all possible cluster vectors for a given ω is $|C|^{|J|}$, so that an exhaustive $\mathbf{c}(\omega)$ clusterability test for even a single mode vector is also computationally intractable. Chances of ever finding an optimal polynomial time solution to the software meta-control problem are very slim, since the problem is readily shown to be equivalent to a composite knapsacking and binpacking problem, and therefore NP-hard. Seeking an optimal mode vector for a single processor is a generalization of the NP-hard zero-one multiple-choice *knapsack* problem, in which rewards and bandwidths are not confined to the integer domain. Verifying whether a given mode vector is clusterable is a generalization of the strictly NP-hard *binpacking* problem, in which bin (processor) capacities need not be uniform and the number of bins is limited. *Optimal* meta-control is therefore computationally intractable even for relatively small sets of jobs and processors. We have therefore devised two practical *approximation* algorithms, QDP and G^2, which are capable of integrated binpacking (clustering) and knapsacking (mode selection) in very reasonable polynomial time.

6.1.5 Evaluating Policy Performance

An important feature of both approximation algorithms is that they also provide us with the means to evaluate the performance of their decisions. Let π denote a given meta-control policy, then the performance, ψ_π, of policy π in a given system state is measured by

$$\psi_\pi = \frac{v(\omega_\pi)}{v^*}, \tag{6.11}$$

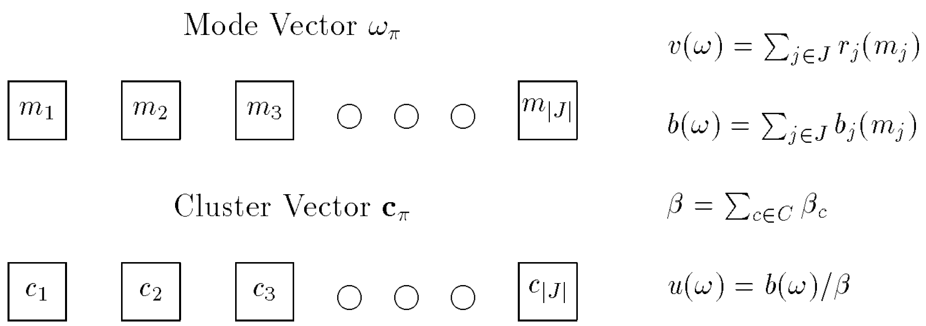

Figure 6.1: Meta-Controller Output

where ω_π is the mode vector produced by the policy π for that state, and $v^* = v(\omega^*)$ is the optimal system gain for that state, as defined by Equation (6.10). Unfortunately, v^* is intractable, but each of our approximation algorithms provides a v^* upper-bound, $\bar{v} \geq v^*$. This provides us with a lower-bound for performance, $\bar{\psi}_\pi$, which can serve as a somewhat pessimistic measure:

$$\bar{\psi}_\pi = \frac{v(\omega_\pi)}{\bar{v}}. \tag{6.12}$$

Note that we later find cases where $\bar{v}$ can be shown to be a very tight upper-bound for v^*, and therefore a very *realistic* measure.

When dealing with dynamic systems, a more relevant performance measure might be the expected *mean* policy performance, $\mathrm{E}\{\psi_\pi\}$, for the anticipated set of system states, whereupon the corresponding pessimistic lower bound measure of mean policy performance is denoted by $\mathrm{E}\{\bar{\psi}_\pi\}$.

6.1.6 Meta-Controller I/O

As summarized in Figure 6.1, the mandatory *output* for any meta-control policy, π, must include a selection of modes, recorded in the policy mode vector, ω_π, as well as a clustering of jobs over processors, recorded in the policy cluster vector, $\mathbf{c}_\pi$. Whenever possible, a policy might also provide a v^* upper-bound, $\bar{v}_\pi$, capable of serving as a pessimistic measure of policy performance.

The mandatory *input* for any meta-control policy must include the current system state, $s = \{\mathcal{J}, \mathcal{C}\}$. Throughout this chapter we will assume that s is provided in the following manner. Any mode $m \in M_j$ is *dominated*

k	$\mathbf{J}_1$			$\mathbf{J}_2$			$\mathbf{J}_3$		
tuple	band	reward	mode	band	reward	mode	band	reward	mode
1	0.6	0.8	1	0.8	0.8	1	0.9	0.4	2
2	0.2	0.3	3	0.5	0.3	2	0.0	0.0	4
3	0.1	0.2	4	0.2	0.2	3			

Table 6.1: Sample Job Lists

by another mode $\hat{m} \in M_j$ if $r_j(\hat{m}) > r_j(m)$ and $b_j(\hat{m}) \leq b_j(m)$, i.e. mode m should never take part in a meta-control mode selection since m can always be replaced by $\hat{m}$ in ω_π without altering $\mathbf{c}_\pi$ to obtain a better system gain. From a meta-control perspective, dominated modes can therefore be eliminated from $\mathcal{J}$. Thus each job j can be fully described by a job parameter list $\mathbf{J}_j = \{\mathbf{J}_j^k\}$ of tuples, where tuple $\mathbf{J}_j^k = \{\mathsf{band}, \mathsf{reward}, \mathsf{mode}\}$, $k = 1 \cdots \mathsf{ntuples}(\mathbf{J}_j)$, $\mathsf{ntuples}(\mathbf{J}_j)$ denotes the number of tuples in $\mathbf{J}_j$ for job j, and the mode value in each $\mathbf{J}_j^k$ tuple identifies a unique *non-dominated* job mode, $m \in M_j$, which would contribute a $\mathsf{reward} = r_j(m)$ to the total system gain if provided $\mathsf{band} = b_j(m)$ in the current system state. All tuples in each job parameter list can therefore be *strictly* monotonic. ordered by decreasing bandwidth and reward values[1]. The application state, $\mathcal{J}$, is therefore described by a full set of job parameter lists, i.e. $\mathcal{J} = \{\mathbf{J}_j\}$, while the platform state, $\mathcal{C}$, is described by a list of processor capacities, $\{\beta_c\}$. We use njobs, ncpus, $\mathsf{ntuples}(\mathbf{J}_j)$, and $\mathsf{nmodes} = \sum_j \mathsf{ntuples}(\mathbf{J}_j)$ to denote the respective cardinalities of the job set J, the processor set C, the number of tuples in the job parameter list, $\mathbf{J}_j$, and the total number of tuples in $\mathcal{J}$. We also use $\beta = \sum_C \beta_c$ to denote the total platform capacity.

Table 6.1 contains a sample $\mathcal{J}$ with $\mathsf{njobs} = 3$. As illustrated, each job parameter list, $\mathbf{J}_j$, contains only those modes which are not dominated, and the strictly monotonic list of tuples is ordered by decreasing bandwidths and rewards. Mode bandwidths and processor capacities are normalized by $\max(\beta_c)$. Mode rewards are also normalized such that all $b_j(m)$ and $r_j(m)$ values in $\mathcal{J}$, here, are within $[0, 1]$. The bandwidth and reward of tuple $\mathbf{J}_j^k$ is denoted by $\mathbf{J}_j^k.\mathsf{band}$ and $\mathbf{J}_j^k.\mathsf{reward}$, e.g. $\mathbf{J}_3^1.\mathsf{reward} = 0.4$ and $\mathbf{J}_3^2.\mathsf{band} = 0.0$. Note that tuple $\mathbf{J}_3^2$ essentially represents a suspension mode for the non-critical Job 3. We also see that $\mathsf{ntuples}(\mathbf{J}_3) = 2$ in this example.

Let ω_0 denote a *minimal* bandwidth mode vector, then $\omega_0 = \{3, 3, 2\}$ in

[1]Note that this monotonic ordering of tuples is not an inherent requirement for any of our algorithms, but it simplifies their implementation.

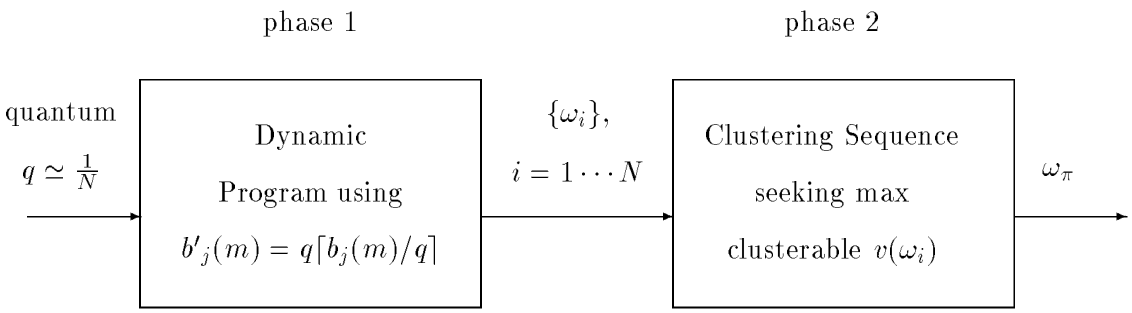

Figure 6.2: Algorithm QDP Diagram

the above example, with $b(\omega_0) = 0.3$ and $v(\omega_0) = 0.4$.

6.2 Algorithm QDP

As illustrated by Figure 6.2, algorithm QDP produces a near-optimal software meta-control decision in two distinct phases. In the first phase, QDP uses *dynamic programming* to simultaneously produce an initial subset $\Omega_\pi \subset \Omega$ of mode vector solutions, where $\Omega_\pi = \{\omega_n\}, n = 0, 1, \cdots, N$. Bandwidth *quantization*, i.e. the rounding up of all bandwidths to the nearest multiple of a given *quantum* input, $q \in \mathcal{R}^+$, is used within this phase to ensure that the number of mode vectors $N = |\Omega_\pi| \sim \beta/q$ will not exceed a given size. The dynamic programming phase is designed to guarantee that $v(\omega_n)$ is an ϵ-approximation of $v^*(\Omega, n \cdot q)$, and it is readily shown that smaller values of ϵ can be obtained by reducing the quantum, q. These mode vectors can then be tested in the second QDP phase for clusterability by any conventional clustering policy, whereupon the most valuable clusterable mode vector is adopted, as described in subsequent subsections. Higher QDP gains can be traded off for shorter response times by controlling the bandwidth quantum q in the first phase and by using appropriate *mode vector sequencing* policies to decide which mode vectors should be tested for clusterability in the second phase.

QDP details are provided by Figure 6.3. As shown, QDP inputs include the system state, $\mathcal{J}$ and $\mathcal{C}$, along with the bandwidth quantum, q, which breaks up the total bandwidth range of $(0, \beta)$ into $\lceil \beta/q \rceil$ discrete band-

```
procedure QDP(input q, 𝒥, 𝒞; output ω_π, c_π, v̄);
var 𝒥̂[nmodes], DP[njobs][S];
begin
        Quantize(𝒥, q, 𝒥̂, B_ε);
        N := ⌈(β + B_ε)/q⌉;
        DynamicProgram(𝒥̂, q, N, DP);
        v̄ := DP.sum_1^S;
        for state := N downto 1 do begin
            GetModeVector(state, DP, ω_π);
            if Clusterable(ω_π, 𝒞, c_π) then return;
            end;
end;
```

Figure 6.3: Algorithm QDP Program

k	$\mathbf{J}_1$			$\hat{\mathbf{J}}_1$			
tuple	band	reward	mode	band	reward	fragment	tuple
1	0.6	0.8	1	0.6	0.8	0.0	1
2	0.2	0.3	3	0.3	0.3	0.1	2
3	0.1	0.2	4	(0.3)	(0.2)	0.2	3

Table 6.2: Sample Quantizations for $q = 0.3$

width values. An appropriate q input is determined by the desired level of first-phase ϵ-optimality or by other meta-controller constraints, e.g. meta-controller resources and response times. As already described in Section 6.1.6, QDP output comprises ω_π, $\mathbf{c}_\pi$, and $\bar{v}$. The mode vector, ω_π, provides the mode selection decision, i.e. $\mathbf{J}_j^{\omega_\pi[j]}$.mode is the mode selected for each job j. The *cluster vector*, $\mathbf{c}_\pi$, provides the valid clustering solution for that decision, i.e. $\mathbf{c}_\pi[j]$ is the processor assigned to each job j. $\bar{v}$ is an ϵ-approximate $v^*(\Omega, \beta)$ upper-bound for the single processor case. The optimal multiprocessor gain cannot exceed that of the optimal uniprocessor gain, so $\bar{v}$ serves also as a pessimistic performance scale for evaluating $v_\pi = v(\omega_\pi)$, as previously described in Section 6.1.3.

6.2.1 QDP Phase 1

Referring again to Figure 6.3, the first QDP phase is accomplished by activating procedure $\mathsf{Quantize}$, computing N, and then calling $\mathsf{DynamicProgram}$. $\mathsf{Quantize}$ translates all $\mathcal{J}$ tuples into $\hat{\mathcal{J}}$ tuples with quantized bandwidths which later take part in the dynamic programming. Each $\mathbf{J}_j^k$ tuple is converted into a extended tuple $\hat{\mathbf{J}}_j^k = \{\mathsf{reward}, \mathsf{band}, \mathsf{fragment}, \mathsf{tuple}\}$ where reward remains the same; band is rounded up to the nearest multiple of q, i.e. $\hat{\mathbf{J}}_j^k.\mathsf{band} = q \cdot \lceil \mathbf{J}_j^k.\mathsf{band}/q \rceil$; $\mathsf{fragment}$ records the internal bandwidth fragmentation, $\hat{\mathbf{J}}_j^k.\mathsf{band} - \mathbf{J}_j^k.\mathsf{band}$; and tuple retains the k index of the original $\mathbf{J}_j$ tuple. If $\hat{\mathcal{J}}$ contains several tuples with equal bandwidths then those with lower rewards are eliminated. Table 6.2 shows the $\hat{\mathbf{J}}_1$ tuples which would be obtained from the $\mathbf{J}_1$ tuples of Table 6.1. Quantized tuples are provided for $q = 0.3$. We can see that all $\hat{\mathbf{J}}_1$ bandwidths are multiples of q and that $\mathsf{fragment}$ in each tuple represents the internal bandwidth fragmentation. We also see how bandwidth quantization has essentially created two tuples with equivalent 0.3 bandwidths, so the lower reward tuple (in parenthesis) was subsequently eliminated. Finally, $\mathsf{Quantize}$ uses the $\mathsf{fragment}$ values in each $\hat{\mathcal{J}}$ tuple to compute B_ϵ by

$$B_\epsilon = \sum_{j=1}^{\mathsf{njobs}} \max_k \hat{\mathbf{J}}_j^k.\mathsf{fragment}. \tag{6.13}$$

Thus B_ϵ sums up the worst-case total fragmentation for any possible mode vector over $\hat{\mathbf{J}}$, and it is returned as output. $\mathsf{Quantize}$ is then followed by the selection of N which dictates the number of Ω_π mode vectors to be produced by dynamic programming and which therefore determines the $\mathsf{DynamicProgram}$ time/space complexity. As we will show, each dynamic programming state, $n = 0, 1, \cdots N$, represents a total bandwidth allocation of $n \cdot q$. A choice of $N = \beta/q$ would ensure that $\mathsf{DynamicProgram}$ will consider all mode vectors subject to $b(\omega) \leq \beta$ without considering losses due to internal fragmentation. By setting

$$N = \lceil (\beta + B_\epsilon)/q \rceil, \tag{6.14}$$

we essentially offset the maximum internal fragmentation B_ϵ to guarantee that *all* relevant mode vectors will be considered. $\mathsf{DynamicProgram}$ may therefore produce mode vectors with total bandwidths which may exceed β

by up to B_ϵ, but these will be immediately disqualified in the second QDP phase which tests for clusterability[2].

The DynamicProgram routine of Figure 6.4 receives $\hat{\mathcal{J}}$, q, and N as input and generates, as output, the initial mode vector subset Ω_π of ϵ-approximate knapsacking solutions which are contained in the internal DP structures, sum, tuple, and state. Each element sum_j^n in matrix DP.sum represents the maximum sum of rewards which can be obtained for jobs $j \cdots \mathsf{njobs}$ with a total quantized bandwidth which does not exceed $n \cdot q$. As we can see, sum_j^n is computed recursively by

$$\mathsf{sum}_j^n = \max_m \{\mathsf{sum}_{j+1}^{n-\frac{\hat{\mathbf{J}}_j^m.\mathsf{band}}{q}} + \hat{\mathbf{J}}_j^m.\mathsf{reward} \mid n \geq \frac{\hat{\mathbf{J}}_j^m.\mathsf{band}}{q}\}, \tag{6.15}$$

where $\forall j > \mathsf{njobs} : \mathsf{sum}_{j+1}^m = 0$. When DynamicProgram terminates, sum_1^n represents the maximum system gain attainable over $\hat{\mathcal{J}}$ for any uniprocessor capacity $n \cdot q$. Dynamic Program also produces the corresponding tuple_j^n and state_j^n matrix elements in DP which record the $\hat{\mathbf{J}}_j^m$ tuple index, m, and the $\mathsf{sum}_{j+1}^{n'}$ state, n', which produced each sum_j^n. As depicted by the GetModeVector procedure in Figure 6.4, these internal tuple and state matrices enable us to reconstruct the ω_n mode vectors which produced each sum_1^n system gain. Clearly this is done in $O(\mathsf{njobs})$ time complexity. Thus DynamicProgram has essentially produced a set Ω_π of ω_n mode vectors, where each ω_n has a system gain, $v_n = \mathsf{sum}_1^n$, with the following property:

$$v^*(\Omega, n \cdot q) \geq v_n = \mathsf{sum}_1^n \geq v^*(\Omega, n \cdot q - B_\epsilon). \tag{6.16}$$

This property follows from Equation (6.8), Equation (6.13), and Equation (6.15) since $n \cdot q \geq b(\omega_n) \geq n \cdot q - B_\epsilon$. Incorporating Equation (6.14) into Equation (6.16), we deduce

$$\bar{v} = v_N \geq v^*(\Omega, \beta), \tag{6.17}$$

so that $\bar{v}$ does indeed constitute an upper-bound for the optimal uniprocessor system gain. Furthermore, from Equation (6.13) and Equation (6.16):

$$\lim_{q \to 0} \bar{v} \to v^*(\Omega, \beta), \tag{6.18}$$

since B_ϵ converges with q to zero, so that QDP is indeed ϵ-approximate for a uniprocessor.

[2]♣◇♣ Verify algorithms with RTSJ paper

```
procedure DynamicProgram( input Ĵ, q, N; output DP );
begin
      for j := njobs downto 1 do
          for n := 1 to N do begin
              DP.sum_j^n := 0;
              for k := ntuples(Ĵ_j)) downto 1 do begin
                  m := Ĵ_j^k.band/q;
                  r := Ĵ_j^k.reward;
                  if j < njobs then r := r + DP.sum_{j+1}^{n-m};
                  if r > DP.sum_j^n then begin
                     DP.sum_j^n := r;
                     DP.tuple_j^n := Ĵ_j^k.tuple;
                     DP.state_j^n := n - m;
                     end;
                  end;
              end;
end;

procedure GetModeVector( input n, DP; output ω );
begin
      for j := 1 to njobs do begin
          ω[j] := DP.tuple_j^n;
          n := DP.state_j^n;
          end;
end;
```

Figure 6.4: Bin Packing and Dynamic Programming

Quantize's time complexity does not exceed $O(\textsf{nmodes})$. DynamicProgram time complexity is $O(\textsf{nmodes}/q)$, since no more than $N = O(1/q)$ states must be considered for each of the $O(\textsf{nmodes})$ tuples in $\hat{\mathcal{J}}$.

6.2.2 QDP Phase 2

This brings us to the second QDP phase where $\omega_n \in \Omega_\pi$ mode vectors must be tested for clusterability over $\mathcal{C}$ to determine the most valuable clusterable mode vector in Ω_π. The QDP version in Figure 6.3 uses *linear sequencing*, a very simple, brute-force, method for doing this. In this method, these mode vectors are tested in decreasing order of n, starting with $n = N$, until a clusterability test succeeds. The procedures GetModeVector and Clusterable are used for reconstructing ω_n and for testing its clusterability. Each ω_n reconstruction uses the original (nonquantized) bandwidth values in $\mathcal{J}$, which are readily obtained via $\hat{\textsf{J}}$.tuple. It follows from Equation (6.8) that $v_n = v(\omega_n)$ is monotonically non-decreasing over n, so that the first clustering test to succeed is indeed the most valuable $\omega_n \in \Omega_\pi$ which is clusterable by whatever clustering policy, γ, is employed by Clusterable. Clustering success is guaranteed when $b(\omega_n)/\beta \leq \bar{u} \geq 0.5$, (see Section 5.1.3) so the worst-case number of tests is $(1 - \bar{u}) \cdot N$. Clusterable details are not provided to emphasize that any clustering policy from Chapter 5 is acceptable. Assuming γ =FFD, then we would have an FFD time complexity of $O(\textsf{njobs}(\textsf{ncpus} + \log \textsf{njobs}))$. With a worst-case number of $(1 - \bar{u}) \cdot N$ clustering tests, the timing complexity of the second phase is $O([\textsf{ncpus} + \log \textsf{njobs}] \cdot \textsf{njobs}/q)$. We therefore summarize that QDP has a total time complexity of $O([\textsf{nmodes}/\textsf{njobs} + \textsf{ncpus} + \log \textsf{njobs}] \cdot \textsf{njobs} \cdot \textsf{ncpus}/q)$. We can generally assume $O(\textsf{nmodes}) \sim O(\textsf{njobs})$ for large njobs, whereupon QDP's time complexity becomes $O([\textsf{ncpus} + \log \textsf{njobs}] \cdot \textsf{njobs} \cdot \textsf{ncpus}/q)$. In Section 6.2.2 we show that there are more cost-effective methods for carrying out the second QDP phase, thereby further reducing QDP time complexity with negligible loss of performance. We also show that the number of clustering attempts is typically only a fraction of N. Note that clustering attempts are necessary for single cluster systems as well, except that we need only verify $b(\omega_n) \leq \beta$. We also need not retest duplicate ω_n mode vectors which can appear for any number of consecutive n.

6.3 Algorithm $\mathbf{G}^2$

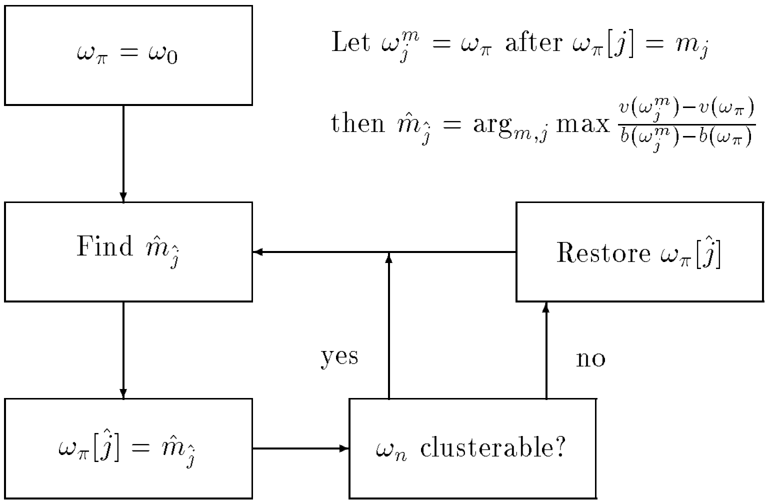

Figure 6.5: Algorithm G^2 Diagram

As indicated by Figure 6.5, G^2 is an iterative algorithm, where an ω_π mode vector is initially set to the minimal bandwidth mode vector, ω_0, and then repeatedly improved by a greedy heuristic at each iteration. Only one mode (tuple) in the ω_π mode vector can be exchanged at each iteration. All $\mathbf{J}_j^k$ tuples with rewards/bandwidths which exceed those already selected in ω_π, i.e. satisfying $k < \omega_\pi[j]$, are referred to as *candidate* tuples which are considered for exchange. Using ω_j^k to denote the mode vector which would be obtained by exchanging mode $\omega_\pi[j]$ with tuple $\mathbf{J}_j^k$, the highest priority tuple for exchange, $\mathbf{J}_{j*}^{k*}$, is that which would provide the greatest reward/bandwidth *gradient*, i.e.

$$j^*, k^* = \arg_{j,k} \max_{j,k<\omega[j]} \frac{v(\omega_j^k) - v(\omega_\pi)}{b(\omega_j^k) - b(\omega_\pi)} \quad . \tag{6.19}$$

Hence the *gradient greedy* characterization. All candidate $\mathcal{J}$ tuples are stored in a priority queue with the highest gradient tuple always being at the top of the list. Each iteration removes the highest gradient candidate in the list and attempts to incorporate it into ω by making an appropriate tuple exchange. The subsequent mode vector is adopted as the following iteration's ω_π only if it is found to be *clusterable*, whereupon the list of candidates must also be appropriately updated. Each successful tuple exchange increases both $v(\omega_\pi)$ and $b(\omega_\pi)$, while shortening the list of candidates. The rate of successful exchanges will naturally decline as $b(\omega_\pi)$ reaches *saturation*, i.e. approaches β. Iterations normally end when the list of candidates is exhausted.

A two-phased G^2 variation appears in Figure 6.6, in which a mandatory first phase produces the software meta-control decision, and an optional second phase produces a $\bar{v}$ upper-bound for estimating the performance of the first phase decision, as previously described in Section 6.1.3. The current system state is provided by $\mathcal{J}$ and $\mathcal{C}$ which are inputs to G^2. As in QDP, the essential G^2 outputs are ω_π and $\mathbf{c}_\pi$ which are produced by the mandatory first phase. $\bar{v}$ is an output provided by the optional second phase, in which gradient greedy iterations are carried out on the $\bar{\omega}$ mode vector, and the final $v(\bar{\omega})$ is adopted as $\bar{v}$. Candidates is a priority queue of tuple candidates ordered by gradient, and it is initialized by Candidates.Init. Job tuples in $\mathbf{J}_j$ are ordered by decreasing bandwidth and reward, so ω_π is initialized as a minimal bandwidth mode vector by selecting the last tuple number, ntuples($\mathbf{J}_j$). AppendJobGradients($\mathcal{J}, \omega_\pi, j$) is used to insert into Candidates all candidate job j tuples, i.e. all $\mathbf{J}_j^k$ tuples subject to $k < \omega_\pi[j]$. We then enter the first phase **while** loop which ends when the candidates list is exhausted. Each iteration uses Candidates.Remove to obtain the j^*, k^* indices of the highest gradient tuple, and the tuple is exchanged with $\omega_\pi[j^*]$ after the latter is recorded in oldtuple for restoration in case clustering fails. As in QDP, Clusterable tests ω_π for clustering over $\mathcal{C}$ using an arbitrary clustering policy, γ, returning valid clustering assignments in $\mathbf{c}_\pi$ if successful. The new mode vector is adopted if clustering succeeds, whereupon Candidates is updated by first using DeleteJobGradients to remove all job j tuples in Candidates and then using AppendJobGradients to reinsert candidate tuples for job j^* with their new gradients. All other non-j^* job gradients are not affected and need not be recomputed. If clustering fails, then $\omega_\pi[j^*]$ is restored with oldtuple. If this is the *first* failure, then ω_π is copied into $\bar{\omega}$ to serve as the initial second phase mode vector, thereby avoiding unnecessary repetition of all earlier iterations in the second phase. The first phase ends with a software meta-control decision consisting of the last ω_π mode selection which found a valid clustering $\mathbf{c}_\pi$.

The optional second phase operates on the $\bar{\omega}$ inherited from the first phase. Second phase iterations end when the list of candidates is exhausted or as soon as $\bar{\omega}$ reaches saturation, whereupon $\bar{v} = v(\bar{\omega})$ is indeed a $v^*(\Omega, \beta)$ upper-bound, having incorporated into it only the highest reward/bandwidths gradient available in $\mathcal{J}$. Using the same argument, we contend that every $\bar{\omega}$

```
var Candidates;
procedure GradientGreedy(input 𝒥, 𝒞; output ω_π, c_π, v̄);
begin
        Candidates.Init;
        for j := 1 to njobs do begin
             ω_π[j] := ω̄[j] := ntuples(J_j);
             AppendJobGradients(𝒥, ω_π, j);
             end;
        while Candidates.NotEmpty do begin
             Candidates.Remove(j*, k*);
             oldtuple := ω_π[j*]; ω_π[j*] := k*;
             if Clusterable(ω_π, 𝒞, c_π) then begin
                DeleteJobGradients(𝒥, ω_π, j*);
                AppendJobGradients(𝒥, ω_π, j*);
                end
             else begin
                if ω̄ = NIL then ω̄ = ω_π;
                ω_π[j*] := oldtuple;
                end;
             end;
        /* Optional v̄ Second Phase: */
        if b(ω̄) < β then begin
             Candidates.Init;
             for j := 1 to njobs do AppendJobGradients(𝒥, ω̄, j);
             end;
        while Candidates.NotEmpty and b(ω̄) < β
             do begin Candidates.Remove(j*, k*);
             ω̄[j*] := k*;
             DeleteJobGradients(𝒥, j*);
             AppendJobGradients(𝒥, j*);
             end;
        v̄ := v(ω̄);
end;
```

Figure 6.6: Two-Phased Algorithm G^2

k	$\mathbf{J}_1$		$\mathbf{J}_2$		$\mathbf{J}_3$	
mode	band	reward	band	reward	band	reward
1	0.6	0.8	0.8	0.8	0.9	0.4
2	0.2	0.3	0.5	0.3	0.0	0.0
3	0.1	0.2	0.2	0.2		

Table 6.3: G^2 Example

	ω_i				
i	$\omega_i[1]$	$\omega_i[2]$	$\omega_i[3]$	$b(\omega_i)$	$v(\omega_i)$
0	3	3	2	0.3	0.4
1	1	3	2	0.8	1.0
2	1	1	2	1.4	1.6

Table 6.4: G^2 Iterations

mode vector has the following optimality property: $v(\bar{\omega}) = v^*(\Omega, b(\bar{\omega}))$. The second phase first checks whether the inherited $\bar{\omega}$ is not already saturated, whereupon it uses Candidates.init and AppendJobGradients to initialize and rebuild a list of Candidates only if it is not saturated. This is followed by the simpler second phase **while** loop which ends as soon as $\bar{\omega}$ saturates, so that only successful tuple exchanges need be considered. We note that, unlike QDP, the $\bar{v}$ upper-bound produced is not ϵ-approximate for a uniprocessor.

As we can see, both phases are very similar in nature, starting with minimal bandwidth mode vectors which eventually reach saturation after a sequence of gradient greedy iterations. They differ in that a first phase tuple exchange is adopted if the subsequent mode vector is clusterable over $\mathcal{C}$, while second phase clustering requires only that the total bandwidth of the subsequent mode vector should not exceed the total capacity β. The second phase is also unique in that iterations end with the first mode vector that fails to cluster, whereas in the first phase, we restore the mode vector and try again with the next best candidate. Earlier gradient greedy iterations are usually identical in both phases. This property is exploited in the algorithm by ensuring that the initial second-phase mode vector is set to the last first-phase mode vector which satisfies the criteria of both phases.

Table 6.4 shows the ω_π mode vectors produced by G^2 for a given $\mathcal{J}$ state shown in Table 6.3, with njobs = 3. We use ω_i to denote the i-th version of ω_π with each successful tuple exchange. We also use ∇_j^k to denote the current gradient for tuple $\mathbf{J}_j^k$. The initial mode vector ω_0 uses the minimal bandwidth modes for each job, $\{3,3,2\}$. The list of candidates will initially contain the following gradients: $\nabla_1^1 = 6/5$, $\nabla_2^1 = 6/6$, $\nabla_1^2 = 1/1$, $\nabla_3^1 = 4/9$, and $\nabla_2^2 = 1/3$. The first G^2 iteration produces $\omega_1 = \{1,3,2\}$, because the highest gradient belongs to $\mathbf{J}_1^1$. We must now remove ∇_1^2 because it corresponds to a mode which has a lower bandwidth than that in ω_1, i.e. $2 > \omega_1[1] = 1$. All other tuples remain the same. The second iteration produces $\omega_2 = \{1,1,2\}$, because the highest gradient belongs now to $\mathbf{J}_2^1$. For $\beta = 1$, the clustering for ω_2 would fail and we would remain with $\omega_\pi = \omega_1$. Consequently, we record $\bar{\omega} = \omega_2$. The only remaining heap tuples are ∇_3^1 and ∇_2^2, which also fail to cluster due to excess total bandwidth. G^2 ends when Candidates is empty, thereby producing $\omega_\pi = \omega_2$, $v(\omega_\pi) = 1.0$ as a near-optimal solution and also providing $\bar{v} = 1.6$ as an upper-bound for $v^*(\Omega, \beta)$. The optimal solution for this case is actually $v^* = v(\{2,1,2\}) = 1.1$.

The gradient-ordered list of Candidates is most efficiently implemented as an *indexed heap*. Each candidate $\mathcal{J}$ tuple is represented by a heap record which contains the tuple's current gradient and identity. The worst-case time complexity of each AppendJobGradients insertion is $O(\log \mathsf{nmodes})$. The heap *index* provides random access to the heap record belonging to any $\mathbf{J}_j^k$ tuple, so that $O(\log \mathsf{nmodes})$ is also the time complexity for each DeleteJobGradients deletion. We assume that the number of tuple candidates for any given job j is $O(\mathsf{nmodes}/\mathsf{njobs})$. The heap size decreases with each iteration, so the maximum number of iterations will never exceed nmodes and actual heap overheads are generally lower. Only one tuple has been exchanged with each clustering attempt, so that clustering overhead is somewhat reduced as well. Assuming FFD clustering, for example, the clustering overhead is only $O(\mathsf{njobs} \cdot \mathsf{ncpus})$, because only one bandwidth needs to be reordered with a negligible timing complexity of $O(\log \mathsf{njobs})$. Consequently, the total G^2 time complexity is $O(\mathsf{nmodes} \cdot \mathsf{njobs} \cdot \mathsf{ncpus} + \mathsf{nmodes}^2/\mathsf{njobs} \cdot \log \mathsf{nmodes})$. We can generally assume $O(\mathsf{nmodes}) \sim O(\mathsf{njobs})$ for large njobs, whereupon G^2 time complexity becomes: $O(\mathsf{njobs}^2 \cdot \mathsf{ncpus})$. Candidates's space complexity is only $O(\mathsf{nmodes})$.

6.4 Monte Carlo Analysis

Having formulated the QDP and G^2 algorithms, we set out to use Monte Carlo analysis to evaluate performance and time complexity. Each Monte Carlo experiment consists of a sequence of 50 random system states and then recording the $\bar{v}$ and ω_π outputs produced by each of several software meta-controller policies for each system state. Each system state is fully defined by an application state, $\mathcal{J}$, and by a platform state, $\mathcal{C}$, where the former comprises of a given number of $\{\mathsf{band}, \mathsf{reward}\}$ pairs for a given number of jobs, and the latter comprises of a list of capacities $\{\beta_c\}$ for a given number of processors. State generation parameters included the number of jobs, the number of modes per job, the number of processors, maximum band and reward values for each mode, minimal β_c capacities for each processor, and the percentage of non-critical jobs which can be shed. Bandwidths, rewards and capacities are generated randomly with a uniform distribution within their prescribed ranges. Bandwidths and rewards for a given job are then sorted and paired to create $\mathbf{J}_j$ tuples which are strictly monotonic. The policies evaluated always included G^2 and several instances of QDP with varying quantum, q, as well as a density-greedy variation of G^2, henceforth referred to as DG. When testing for small numbers of jobs, modes, and processors, an exhaustive combinatoric search is used to determine the optimal v^* values for each case, whereupon Equation (6.11) serves to compute actual ψ_π performance for each policy. In all other cases, $\bar{\psi}_\pi$ is computed using the $\bar{v}$ produced by QDP with a very small q, whereupon Equation (6.12) serves to compute $\bar{\psi}_\pi$ lower bounds for policy performance. When evaluating average policy performance, a distinction was also made between different average *loads*, where an average load of ρ means that mode vector utilizations, $u(\omega)$, for mode vector $\omega \in \Omega$, are normally distributed with an average utilization of $\rho = \mathrm{E}\{u(\omega)\}$.

We recall that $\bar{v}$ is an ϵ-approximation of the single processor $v^*(\Omega, \beta)$ optimum which can be significantly lower than the multiple processor optimum, v^*. In Section 7.3, however, we show that $\bar{\psi}_\pi$ lower bounds are actually very tight, i.e. very close to actual ψ_π policy performance, for *large* scale applications with *lower* average loads, e.g. $\rho \simeq 1$. This result is actually quite intuitive, since the clusterability of high-utilization mode vectors is clearly higher when clustering larger numbers of smaller items. Consequently, $\bar{\psi}_\pi$ lower bounds do indeed loosen up, i.e. become more conservative, as applications shrink in size, or when average loads grow larger.

6.4.1 Estimating Policy Performance

Software meta-control is primarily concerned with maintaining near-optimal performance over time, i.e. for a large number of application and platform states. We therefore use Monte Carlo analysis to estimate the *mean* performance $\mathrm{E}\{\Psi_\pi\}$ of a given meta-controller policy π for any given sys-

jobs	load	cpus	modes	q_0	$\mathrm{E}\{\bar{\Psi}_\pi\}$ DG	G^2	QDP_0	QDP_1
6	1.0	3	18	0.012	0.745	0.969	0.969	0.977
			24	0.009	0.632	0.972	0.971	0.978

jobs	load	cpus	modes	q_0	$\mathrm{E}\{\Psi_\pi\}$ DG	G^2	QDP_0	QDP_1
6	1.0	3	18	0.050	0.795	0.986	0.998	0.999
			24	0.050	0.609	0.981	0.989	0.991
5	0.8	3	15	0.050	0.768	0.991	0.997	0.998
			20	0.050	0.681	0.985	0.994	0.997
			25	0.050	0.608	0.983	0.991	0.994
7	1.2	3	21	0.050	0.790	0.983	0.991	0.993
8	1.3	3	24	0.025	0.814	0.985	0.992	0.995

Table 6.5: Precise Performance Measurements

tem, where the random variable Ψ_π represents the expected distribution of randomly sampled ψ_π measurements over time. This distribution reflects the expected distribution of all expected system states, $\mathcal{J} \times \mathcal{C}$. Table 6.5 presents the estimated mean performance, $\mathrm{E}\{\Psi_\pi\}$, for several *smaller* systems obtained in this manner. In each of the applications in this table, the number of jobs, processors, and modes is small enough to enable testing all possible permutations for v^*. Each line in the table corresponds to a single experiment consisting of a sequence of 50 pseudo-random application states with a given average load, for an application with the listed number of jobs, processors, and total number of modes. Results shown are for policies DG and G^2, as well as two instances of QDP: QDP_0 and QDP_1, where QDP_0 uses the listed quantum q_0 computed by

$$q_0 = \mathsf{ncpus}/(10 \cdot \mathsf{nmodes} \cdot \mathsf{njobs}), \tag{6.20}$$

and QDP_1 uses $q_1 = q_0/4$. We see that $\mathrm{E}\{\Psi_\pi\}$ measurements are exceedingly high for these applications, with the QDP policies guaranteeing remarkable performance levels of 0.99 for all cases tested, subject to average load levels which do not exceed 1.0. $O(N) \simeq O(\mathsf{ncpus}/q)$, so we can therefore assume that $O(N) = O(\mathsf{njobs} \cdot \mathsf{nmodes})$ to obtain the QDP time complexity at the end of Section 6.2.

Computational resources were not sufficient enough to compute ψ_π for the larger systems, such as those of Table 6.6. For such systems, we use random variable $\bar{\Psi}_\pi$ to represent sampled $\bar{\psi}_\pi$ measurements, whereupon $E\{\bar{\Psi}_\pi\}$ serves as a lower bound for $E\{\Psi_\pi\}$. Both QDP and G^2 have been designed to produce a $\bar{v}$ upper-bound to facilitate these measurements. For the sake of comparison, Table 6.5 also provides the $E\{\bar{\Psi}_\pi\}$ results for two experiments with 6 jobs, 3 processors, and an average load of 1.0. The first experiment has 3 modes per job for a total of 18 modes, while the second experiment has 4 modes per job. We find that the conservative $E\{\bar{\Psi}_\pi\}$ measurements are 2-3 percent lower than the $E\{\Psi_\pi\}$ measurements for each of the two experiments, as expected for applications of this size.

Table 6.6 provides additional $E\{\bar{\Psi}_\pi\}$ measurements for several larger-scale experiments with 12/24 jobs, 3/5 processors, 4/6 modes per job, and varying average ρ loads. Applications were tested for average loads between 1.0 and 3.0, since higher load ratios are unlikely. We see that G^2's performance is generally 20-40 percent better than that of DG, is comparable to that of QDP_0, and is only a few percent less than that of QDP_1. All G^2 and QDP policies provided performance levels exceeding 0.95 which they only appear to improve for larger numbers of jobs, tuples, and processors. These measurements are also significantly effected by the average *load*, with higher loads reducing performance guarantees by a few percent. As previously mentioned, $\bar{\Psi}_\pi$ lower bounds are much looser for higher loads, so actual performance for higher loads is somewhat better. We also find that G^2 performance is even slightly better than QDP for average loads of 1.0.

6.4.2 Policy Response Times

Table 6.7 presents response times measured for the same DG, G^2 and QDP-based policies. All Monte carlo experiments were carried out on a 486/33SX PC and all measurements are in millisecond units. As indicated, a 24 job application with 4 modes per job, running on a 5 processor platform, with an average mode vector utilization of 1.0, requires an average of 85.9 milliseconds to make an automated meta-control decision using the G^2 algorithm. When using QDP with a quantum $q_0 = 0.062$, such a decision requires an average of 269.2 milliseconds. When using QDP with a more refined quantum $q_1 = q_0/4$, an average decision requires 1.57 seconds, which is substantially slower. Referring back to Table 6.6, we can see that little is gained by a q_1 quantum, since the average performance (see Table 6.6) is less than 1 percent better than that of QDP with quantum q_0, and it is only 0.1 percent

jobs	load	cpus	modes	q_0	$E\{\bar{\Psi}_\pi\}$ DG	G^2	QDP_0	QDP_1
24	1.0	5	96	0.062	0.569	0.989	0.981	0.990
	1.5				0.608	0.983	0.976	0.986
	2.0				0.625	0.978	0.974	0.982
	2.5				0.640	0.976	0.976	0.981
	3.0				0.625	0.969	0.963	0.975
24	1.0	3	96	0.037	0.634	0.977	0.973	0.981
	1.5				0.668	0.970	0.969	0.976
	2.0				0.685	0.963	0.967	0.971
	2.5				0.737	0.964	0.969	0.972
	3.0				0.726	0.960	0.963	0.967
24	1.0	3	144	0.025	0.559	0.980	0.976	0.984
	1.5				0.599	0.978	0.977	0.981
	2.0				0.618	0.975	0.978	0.980
	2.5				0.634	0.974	0.974	0.976
	3.0				0.615	0.973	0.972	0.977
12	1.0	5	48	0.031	0.552	0.989	0.981	0.991
	1.5				0.588	0.982	0.981	0.987
	2.0				0.609	0.978	0.976	0.982
	2.5				0.622	0.975	0.972	0.978
	3.0				0.647	0.966	0.963	0.969
12	1.0	3	48	0.018	0.657	0.974	0.973	0.980
	1.5				0.648	0.973	0.971	0.978
	2.0				0.680	0.968	0.968	0.973
	2.5				0.720	0.961	0.964	0.969
	3.0				0.733	0.956	0.951	0.955
12	1.0	3	72	0.012	0.568	0.978	0.976	0.983
	1.5				0.589	0.978	0.979	0.983
	2.0				0.626	0.977	0.979	0.983
	2.5				0.643	0.975	0.974	0.979
	3.0				0.655	0.972	0.973	0.974

Table 6.6: Conservative Performance Measurements

					E{$msecs$}			
jobs	load	cpus	modes	q_0	DG	G^2	QDP_0	QDP_1
24	1.0	5	96	0.062	142.461	85.859	269.161	1568.321
	1.5				118.758	68.738	259.739	1561.288
	2.0				96.316	63.547	248.482	1555.852
	2.5				91.000	55.422	253.877	1554.127
	3.0				91.783	50.562	256.727	1557.357
12	1.0		48	0.031	144.224	82.351	271.383	1558.804
	1.5				109.182	67.224	249.682	1568.077
	2.0				94.883	58.236	255.837	1556.969
	2.5				100.159	53.659	245.660	1566.186
	3.0				82.382	53.816	252.256	1568.535
24	1.0	3	96	0.037	77.158	47.939	244.434	1530.909
	1.5				72.883	43.048	254.920	1510.063
	2.0				48.938	48.413	242.036	1498.564
	2.5				53.395	40.127	235.796	1490.592
	3.0				51.480	37.685	236.885	1481.895
12	1.0		48	0.018	80.165	48.619	246.296	1530.549
	1.5				72.991	41.025	249.519	1515.960
	2.0				61.178	36.702	242.270	1501.586
	2.5				54.489	40.899	239.961	1493.601
	3.0				51.811	39.458	241.369	1506.125
6	1.0		24	0.009	76.035	49.861	251.749	1528.707
	1.5				59.991	45.824	240.594	1511.523
	2.0				61.172	37.040	243.543	1494.323
	2.5				60.843	41.465	241.650	1522.022
	3.0				54.159	35.481	241.174	1499.026

Table 6.7: Response Time Measurements

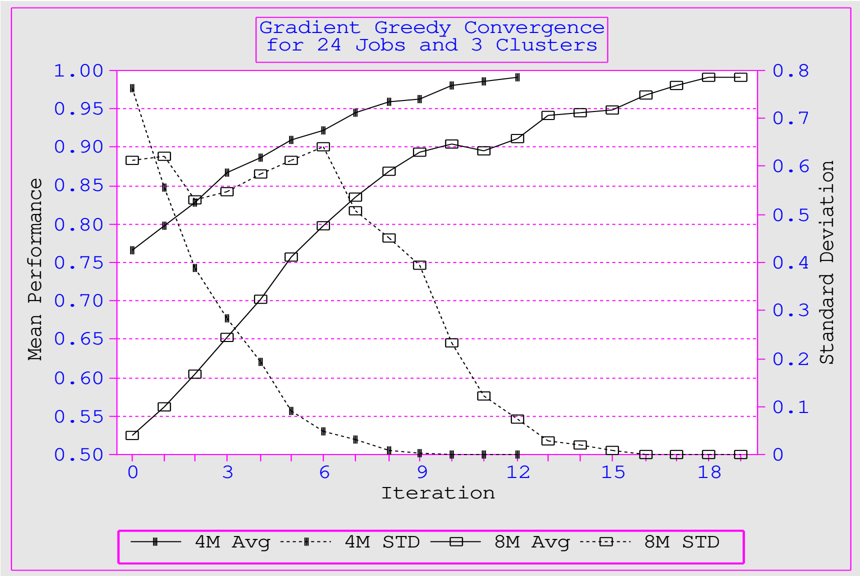

Figure 6.7: G^2 Convergence

better than that of G^2.

We also see that the average load has a relatively small effect on the response times. QDP policy response times are clearly linearly affected by the number of the jobs divided by the quantum. G^2 response times are also several times better than that of QDP_0 despite comparable levels of performance. G^2 response times were also better than that of DG, despite the G^2 need to recalculate gradients with each successful tuple exchange. The reason for this is that the number of iterations is a more significant factor and G^2 saturates more quickly than DG. Another factor is that these complexities are overshadowed by the clustering complexity which dominates. Note that the G^2 and QDP_0 response times are quite reasonable for many applications.

6.4.3 G^2 Convergence

Figure 6.7 illustrates how G^2 converges to its final values. The solid plots show the mean performance obtained for various numbers of iterations for two 50-state experiments. Both experiments had 24 jobs running on 3 processors, with the *4M* experiment having 4 modes per job and *8M* experiment having 8 modes per job. The dotted plots show the standard deviations.

	jobs	load	cpus	modes	q_0	DG	G^2	QDP_0	QDP_1
$\min\{\bar{\Psi}_\pi\}$	24	1.0	5	96	0.062	0.471	0.974	0.953	0.981
$\max\{\bar{\Psi}_\pi\}$						0.745	0.995	0.996	0.996
$E\{\bar{\Psi}_\pi\}$						0.569	0.989	0.981	0.990
$STD\{\bar{\Psi}_\pi\}$						0.066	0.005	0.009	0.004
$\min\{ms\}$						105.271	21.178	239.308	1513.917
$\max\{ms\}$						190.362	133.376	312.379	1637.207
$E\{ms\}$						142.461	85.859	269.161	1568.321
$STD\{ms\}$						22.446	26.522	20.768	29.424

Table 6.8: Performance and Response Time Variance

Worst-case G^2 time complexities assume a maximum of nmodes iterations, but we can see that only a fraction of nmodes is actually required. In the *4M* experiment, for example, nmodes = 96, yet the average number of iterations is only 21.8 with a maximum of 65 and a standard deviation of 18.8. The *8M* experiment has nmodes = 192, but the average number of iterations is 99.9 with a maximum of 149 and a standard deviation of 36.5. G^2 *should* converge quickly, since the rate of successful tuple exchanges should decrease as $b(\omega_\pi)$ reaches *saturation*. G^2 response time can be therefore customized by limiting the number of iterations. As indicated by these graphs, the G^2 response time can thus be significantly reduced with negligible loss of performance. In the *8M* experiment, for example, limiting the number of iterations to 13 would produce a performance of 0.953 instead of 0.982 while reducing the response time by 87 percent. Referring back to Table 6.7, G^2 response times could thus be reduced to less than 10 milliseconds while still guaranteeing very high average degrees of performance.

6.4.4 Policy Variance

Another important factor in evaluating policy performance is variance, which reflects stability. Table 6.8 provides a typical sampling of extreme (min/max) performance and response time measurements, along with averages and standard deviations. Results here are provided for only one experiment with 24 jobs, 5 processors, 96 modes, and an average load of 1. We see that G^2 and QDP performance is exceedingly stable, with standard deviations of less than a percent. This is much better than the conventional 6.6% standard deviations obtained for DG. Response time standard deviations are similar for all policies tested. As already noted in the previous subsection, G^2 can

jobs	load	cpus	modes	q	E$\{\bar{\Psi}_\pi\}$	STD$\{\bar{\Psi}_\pi\}$
24	1.0	3	144	0.050	0.952	0.019
				0.025	0.976	0.009
				0.012	0.983	0.005
				0.006	0.984	0.005
				0.003	0.984	0.004
24	2.0	3	144	0.050	0.962	0.018
				0.025	0.977	0.011
				0.012	0.980	0.009
				0.006	0.981	0.008
				0.003	0.981	0.008
12	2.0	5	36	0.167	0.932	0.031
				0.084	0.962	0.018
				0.042	0.973	0.009
				0.021	0.974	0.009
				0.010	0.975	0.009

Table 6.9: Quantum-Controlled QDP Performance

be terminated after any given iteration, enabling us to reduce the response time variance with very little loss of performance. As described in Section 7.3.1, QDP can be similarly revised to obtain similar results.

6.4.5 Cost-Effective Quantum

Table 6.9 provides performance measurements for a larger set of QDP instances with varying quantum values to illustrate the effect of the quantum, q, on performance. Results are provided for 3 experiments using 5 different q values, and we can see that in each of these experiments, little is gained by reducing q beyond a certain point. Extensive testing has brought us to suggest that Equation (6.20) can be useful in estimating a *cost-effective* q value, since smaller quantum values will generally reap only very small improvements. Thus, q currently has a sub-linear effect on performance. In Section 7.3 we explore this phenomenon and show how it can be improved. We also note that certain levels of fragmentation in the knapsacking phase actually improve clusterability, i.e., what is detrimental for knapsacking is actually a catalyst to binpacking.

6.5 Optimizing QDP Phase 2

We now address the issue of maximizing the *mode vector sequencing* policy of QDP phase 2 (Section 6.2.2) when the response time is limited. QDP provides zero value, i.e. $v_\pi = 0$, if it does not complete the first phase or if it does not find a clusterable mode vector in the second phase within the required response time. First phase execution time is relatively constant for a given bandwidth quantum size, so that the quantum size can be chosen to guarantee first phase completion within the required response time. It is the second phase which is more problematic, because the number of produced mode vectors can be very large, and assuming a $\gamma = FFD$ policy for clustering, then each clusterability test has a time complexity of $O(\mathsf{njobs} \cdot (\mathsf{ncpus} + \log\ \mathsf{njobs}))$. If mode vectors are tested by decreasing value, then the first one to successfully cluster can be immediately adopted. But then the number of preceding failures can be anywhere between zero and a very large number, so that second phase execution time can vary significantly. We could always assume *worst-case* execution times, i.e. assuming that *all* mode vectors with $u(\omega) \leq \bar{u}$ will be tested, but this would significantly degrade the bandwidth quantum size unnecessarily. We therefore seek a method for maximizing QDP value within a reasonably small number of clusterability tests. In this *best-effort* sequencing problem, we assume that each mode vector provided by the first phase has a known value and a known clusterability probability. Given a limited number of clusterability tests, we must decide which of the mode-vector solutions should be tested for clusterability to maximize the expected QDP value. We first derive a Markov decision process model and then derive an optimal solution. We then verify our findings by Monte Carlo analysis to demonstrate that a very small number of clusterability tests is indeed sufficient to obtain a nearly optimal expected gain. Note that throughout this section we use s and j to denote Markov decision process states due to standard conventions, and they are not to be confused with the system state s and the job j symbols used throughout this thesis.

6.5.1 The Markov Decision Process

As described in Section 6.2.1, the first QDP phase produces a set $\Omega_\pi = \{\omega_{(n)}\}$ of mode vectors, $n = 1 \cdots N$. Each ω_m has the property that its capacity utilization, $u(\omega_{(n)})$, is very close to n/N and its gain is an ϵ-approximation for $v^*(\Omega, n/N)$, i.e. nearly optimal for all mode vectors which

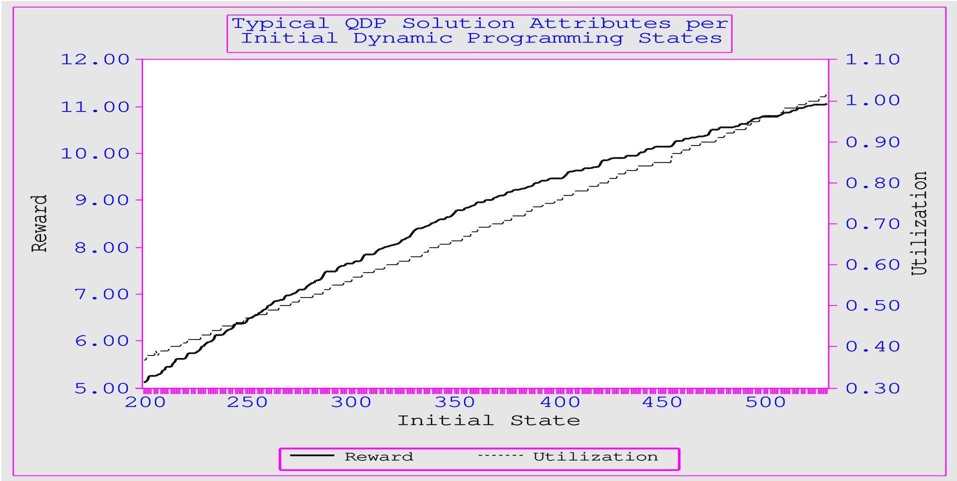

Figure 6.8: Typical QDP Solution Attributes

have a utilization which does not exceed n/N. Thus $\omega_{(n)}$ gains and utilizations are inherently monotonic non-decreasing over Ω_π. Figure 5.1 illustrates how the clusterability, i.e. the probability that $\textsf{clusterable}(\gamma, \omega_{(n)}) = \textsf{true}$, is effected by mode vector utilization. As we can see, the probability of success converges to 1 for utilizations below 0.8, and it goes down to zero as the utilization approaches 1.0.

Each $\omega_{(n)}$ is therefore characterized by a known gain $v_n = v(\omega_{(n)})$, a known utilization $u_n = u(\omega_{(n)})$, and an estimated probability p_n that it will successfully cluster. Figure 6.8 presents typical v_n (**reward**) and u_n (**utilization**) values for a case of $N \simeq 520$, with solutions generated in the first QDP phase for 18 jobs running on 6 processors with 5 modes per jobs. The second QDP phase now seeks the highest gain $\omega_{(n)} \in \Omega_\pi$ which is clusterable. Let n^* denote this highest gain mode vector, i.e.

$$n^* = \arg_n\{\forall n < n^* : \text{clusterable}(\gamma, \omega_{(n)}) = \textsf{false}\} \tag{6.21}$$

What sequence of $\omega_{(n)}$ mode vectors do we test to find n^* within a minimal number of test?

Recalling that all $\omega_{(n)}$ mode vectors with $u(\omega_{(n)}) \leq \bar{u}$ are clusterable, then $n^* \geq \bar{n}$, where $\bar{n} = \lfloor N \cdot (1 - \bar{u}) \rfloor$. Each clustering test is a decision epoch, with $T = \{1, \cdots, M\}$, $M < N - \bar{n}$ being the maximum number of tests which we permit. At each epoch $t \in T$, the current state, $s \in S$, represents the n of the most rewarding solution in Ω_π which has been successfully tested for clusterability prior to t. Consequently, $S = \{\bar{n} \cdots, N\}$, with $\bar{n}$ being the initial state, since ω_{barn} will always cluster. Let Ω_t denote the set of mode vectors in Ω_π which have *not* been tested prior to decision epoch, t, then the available action set A_s for any given state s at epoch t is to end the process or to select a higher gain mode vector $n > s$ to be tested, subject to $\omega_n \in \Omega_t$, i.e. that A_s are actions which has not yet been taken. Let H_t denote the set of actions taken prior to epoch t, then

$$A_s = \{0, s+1, \cdots, N\} \cap \bar{H}_t, \tag{6.22}$$

where 0 here indicates that there is no point in continuing.

The state transition probability for reaching state j when action a is taken in state s is determined by

$$p_t(j \mid s, a) = \begin{cases} p_j, & j = a; \\ 1 - p_j, & j = s; \\ 0, & \text{otherwise}, \end{cases} \tag{6.23}$$

where p_j denotes the probability that $\omega_{(j)}$ will cluster. The subsequent reward, r_t, for epoch t is determined by the the gain of the subsequent state j, $r_t(s, j) = v_j$. The optimality criterion is to maximize the expected value for r_M, which is the gain of the highest gain mode vector which has been found clusterable in the M tests carried out.

6.5.2 Linear Sequencing

For the sake of comparison, we repeat the brute-force *linear sequencing* policy described in Section 6.2.2. Let $\bar{\lambda}$ denote the linear sequencing policy, the the action taken by policy $\bar{\lambda}$ at any given epoch t is

$$a_t^{\bar{\lambda}} = \begin{cases} 0, & s_t \neq 0; \\ N - t + 1, & \text{otherwise}. \end{cases} \tag{6.24}$$

Clearly this policy *must* stop within $N - \bar{n}$ epochs. Figure 6.9 plots the expected reward $\mathrm{E}\{r_M\}$ and standard deviation $\mathrm{STD}\{r_M\}$ as a function of the number of of decision epochs, M, with $N \simeq 520$ for 500 randomly generated Ω_π mode vector sets for 18 jobs running on 6 processors. As indicated, a linear sequencing policy will generally end after only 20 attempts, which is significantly less than N. It will also rarely stop before 7 attempts.

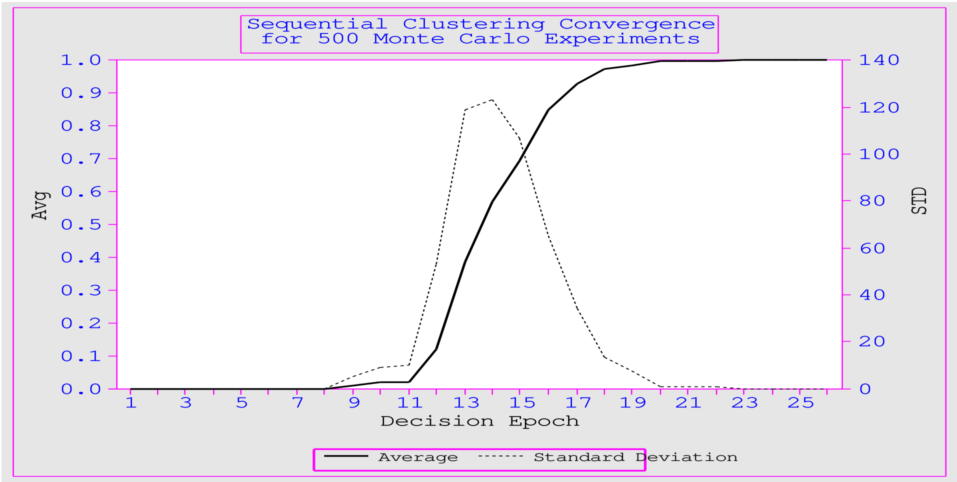

Figure 6.9: Expected Reward for a Linear Sequencing Policy

6.5.3 Gradient Greedy Sequencing

We now seek an optimal *stationary* sequencing policy, i.e. a policy which considers only the current state in making its decisions. We contend that the following *gradient greedy* stationary sequencing policy is optimal for the best-effort sequencing problem, so we use λ^* to denote it:

$$a_t^{\lambda^*} = \begin{cases} 0, & \Omega_t = \phi; \\ \arg\max_{j \in \Omega_t} \{p_j \cdot (v_j - v_s)\}, & \text{otherwise.} \end{cases} \tag{6.25}$$

This is shown to be optimal for maximizing $\mathrm{E}\{r_M\}$ by induction:

$$\begin{aligned} \mathrm{E}\{r_{t+1}\} &= \mathrm{E}\{r_t\} \cdot (1 - p_{a_t}) + v_{a_t} \cdot p_{a_t} \\ &= \mathrm{E}\{r_t\} + (v_{a_t} - \mathrm{E}\{r_t\}) \cdot p_{a_t}. \end{aligned} \tag{6.26}$$

It follows that the way to maximize $\mathrm{E}\{r_{t+1}\}$ is to maximize $\mathrm{E}\{r_t\}$ and to choose an action a_t which will maximize $(v_{a_t} - r_t) \cdot p_{a_t}$. From Equation (6.25), it is obvious that the latter is obtained by adopting policy λ^* at epoch t. It follows by induction that the former is obtained by adopting λ^* for epoch $t-1$. We therefore conclude that a stationary λ^* policy is optimal and that Equation 6.26 can be used to compute the expected gain r_M for any number M of decision epochs.

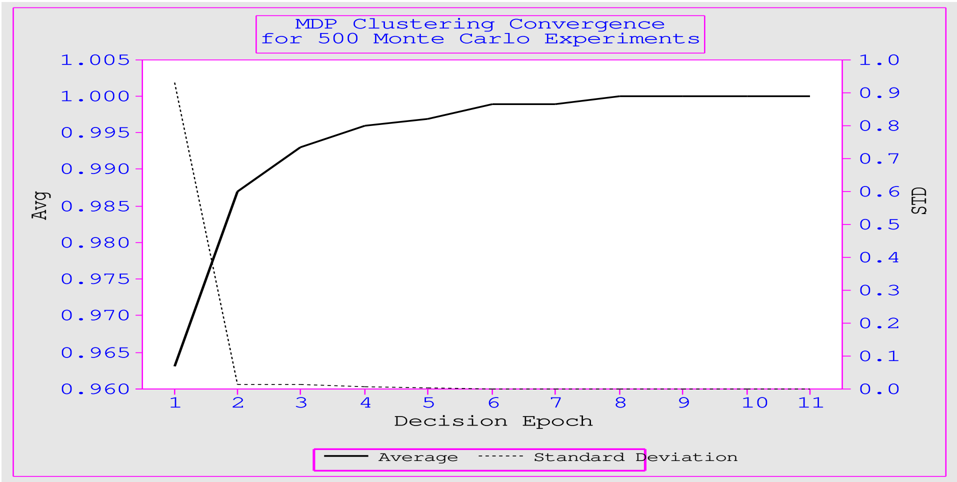

Figure 6.10: Expected Reward for MDP Clustering

6.5.4 Experimental Results

These analytic results were verified by Monte Carlo analysis. The experiment previously used to test $\bar{\lambda}$ in Section 6.5.2 was repeated for λ^*. Results are presented in Figure 6.10, which illustrates the λ^* is indeed a very effective policy in this context. We see that $\mathrm{E}\{r_M\}$ generally converges to the optimal within 8 epochs, already producing nearly optimal rewards after 2 epochs. A nice feature is that it is strictly monotonic, being very useful also when M is not known *a priori*.

6.5.5 Concluding Remarks

Equation 6.26 can be used to estimate the expected gain of a QDP-based policy which allows only M second phase clustering tests. This is accomplished by replacing v_{a_t} with $\mathrm{E}\{v_t\}$ in the equation, and by using Monte Carlo analysis for an adequate number of application states to approximate $\mathrm{E}\{v_t\}$.

Such an approach can also substantially reduce QDP time complexity. Experimental results seem to indicate that adopting $M \sim O(\log N)$ will guarantee a negligible loss of gain while reducing the second phase time complexity by a factor of $\frac{\log N}{N}$. The time complexity of determining $a_t^{\lambda^*}$

in Equation 6.25 is $O(N)$. In Section 6.4.5 we found that a cost-effective bandwidth quantum, q, is such that $O(N) \sim O(\mathsf{njobs} \cdot \mathsf{ncpus})$ so this complexity is usually less than the complexity of the clustering test which follows each decision. Thus these additional λ^* timing complexities can usually be ignored.

We also note the similarity between λ^* and the gradient-greedy G^2 algorithm, and wonder to what extent this might serve an analytic evaluation of G^2.

6.6 Previous Work

As far as we know, our work is also the first to provide approximate solutions to an *integrated* binpacking and knapsacking problem. As already described in Chapter 5, both binpacking (Section 5.1.1) and knapsacking problems (Section 5.2.1) have already been extensively studied and several fast approximation algorithms have been devised for each one of them. To enhance knapsacking performance, more sophisticated polynomial-time schemes have been proposed by Sahni [Sah75] and by Ibarra and Kim [IK75], which guarantee ϵ-approximate returns, i.e. if v^* is the system gain of an optimal solution, and v_π is the gain produced by the algorithm π, then π performance complies to $v_\pi/v^* \geq 1 - \epsilon$. The Ibarra and Kim scheme employs dynamic programming and state elimination by reward scaling, to obtain *fully* polynomial time complexities, bound by polynomial functions of $1/\epsilon$. This approach has also been applied by Lawler [Law77] to the multiple-choice knapsacking variant, which closely resembles our meta-control problem for a single processor. Using the Lawler algorithm, time and space complexities would be $O(\mathsf{nmodes} \cdot \mathsf{njobs}^2/\epsilon)$ if applied to meta-control on a uniprocessor.

The QDP algorithm has essentially extended the Lawler approach to accommodate clustering for multiprocessor platforms, and it supersedes reward (profit) scaling with bandwidth quantization. The primary QDP advantage is that the number of dynamic programming states and maximum clustering attempts is predetermined before dynamic programming begins. It is also designed to yield mode vectors with a wide distribution of utilizations, thereby ensuring that the second QDP phase will always find a *clusterable* mode vector with a near-optimal system gain. An appropriate distribution of utilization also enables the second QDP phase to do so in a cost-effective manner.

G^2 bears a strong resemblance to the well-known DG knapsacking algorithm of Section 5.2.2, differing only in that the DG algorithm, the highest exchange priority is given to tuples with the highest reward/bandwidth ratios which represent *densities* rather than gradients. DG would appear to have a time complexity advantage because tuple densities are fixed and therefore need not be recomputed as ω_π gets transformed. Nevertheless, as described in Section 6.7, G^2 actually does better than DG because the clustering overheads are more dominant and because G^2 convergence is more pronounced.

6.7 Conclusions

Having proposed a fully automated model for software meta-control, this chapter has devised and evaluated two practical families of algorithms, QDP and G^2, for making software meta-control decisions when the system gain function is additive. We have found that *both* families appear to exhibit excellent statistical performance in very reasonable polynomial time. Both families are readily customized to support limited response times, and objective measures have been devised for evaluating their performance. Software meta-controller decisions for large applications can be made in fractions of a second, while producing system gain which is suboptimal by only a few percent. These results clearly indicate the automated software meta-control is a viable alternative for many complex real-time systems.

When comparing the two algorithm families, QDP has a slight gain performance edge over G^2, but G^2 has a lower time/space complexity. Assuming FFD clustering and linear sequencing, QDP time complexity is equal to $O([\mathsf{ncpus} + \log \mathsf{njobs}] \cdot \mathsf{njobs} \cdot \mathsf{ncpus}/q)$ while FFD-based G^2 time complexity is equal to $O(\mathsf{njobs}^2 \cdot \mathsf{ncpus})$. Lower complexities can be obtained by using more efficient clustering and mode vector sequencing policies. Both algorithms appear to perform exceedingly better than a more conventional DG-based approach. A nice feature of QDP is that the quantum q can be chosen to fit available meta-control resources. QDP also provides tighter $\bar{v}$ values which are useful for evaluating performance. G^2 advantages include that it is a quickly converging *monotonic* algorithm, i.e. it can be aborted with a valid solution at any stage.

One might argue that the experimental results presented here might apply only to real-time applications with state distributions similar to those generated in our Monte Carlo experiments. Nevertheless, we contend that consistent behavior over a wide-range of randomly generate system states strongly suggests that they will hold for a wide range of real-time applications. When considering these algorithms for a given application, this can also be verified using the analytic methods of Section 7.3. We also contend that both QDP and G^2 are readily adapted to support additional system gain function forms, but we have yet to determine limitations. The primary QDP restriction is that v^* be definable recursively so that we can apply dynamic programming. The notion of a gradient used by G^2 appears to be even more accommodating.

Future research should determine what kind of system gain functions are most appropriate for various application domains, and how these gain functions might be best accommodated in terms of gain optimality, response times, and the ability to customize these tradeoffs for the specific needs of each application.

Chapter 7

Policy-Making

The *policy-maker* is the top RtTS control layer in Figure 3.2 which decides *when* to activate the meta-controller, which jobs and processors should be considered (*scope*), what algorithms (*methods*) to use, and whether higher optimality (*depth*) should be traded off for quicker response time. This chapter addresses a variety of policy-making issues, such as what is a meta-control policy, how to estimate policy performance for a given application, how to select a QDP-based meta-controller policy to obtain a given level of performance, and how to achieve optimal policy-making when run-time circumstances are known or not a priori known.

Section 7.1 introduces the notion of a meta-control policy. Policy performance for policy π in system state s is defined to be equal to $v_\pi(s)/v^*(s)$, where $v_\pi(s)$ is the system gain produced by policy π for state s, and $v^*(s)$ is the optimal system gain for s. v_π is readily measured, but v^* is computationally intractable, so Section 7.2 develops a statistical model for estimating v^* for any given state. Section 7.3 then extends the statistical model to estimate the v_π gain which should be produced by QDP for any given state as a function of the bandwidth quantum, q. This q was originally introduced to control QDP time complexity by limiting the number of dynamic programming states. The statistical model enables us to estimate also the effect of q on the expected v_π. A novel by-product of this analysis was that QDP was found to be statistically ϵ-approximate for a well-defined range of system states. When attempting to verify this with empirical Monte-Carlo results, we found that this potential was not being realized in our current QDP implementation. Possible remedies are presented in Section 7.3.1.

Section 7.4 then discusses the issue of *optimal* stationary and adaptive policy-making, showing how an optimal policy is readily determined when the distribution of system states, S, can be anticipated. When S is not a priori known, we propose adopting and adapting an adaptive *follow-the-leader* scheme originally devised for Markov renewal programs [FR73], and we use Monte Carlo analysis to evaluate its performance. Section 7.5 concludes with a summary of additional policy-making considerations and offers suggestions for further research in this area.

7.1 Meta-Control Policy

Chapter 6 introduced two meta-control algorithms, QDP and G^2, and demonstrates how both of these algorithms provide appropriate mechanisms for trading off time and performance. Also presented are several variants for each algorithm. QDP, for example, has two proposed methods for carrying out bandwidth quantization and it can use any of several second-phase sequencing algorithms for seeking the highest gain mode vector which is clusterable. Both QDP and G^2 can employ any of several clustering algorithms for testing the clusterability of a given mode vector. There are also several different variations for completing the decision-making process, e.g. an expired maximum execution time, a maximum number of iterations, or simply having reached the best possible alternative offered by the algorithm. We therefore prefer to treat each of these QDP and G^2 algorithms as a *family* of meta-control policies. The QDP family comprises an infinite number of possible decision making policies, each policy being characterized by a specific q quantum value, a completion criteria, and specific algorithms for quantization (Section 6.2.1), for clustering (Chapter 5), and for sequencing in seeking the highest gain mode vector which is clusterable (Section 6.5). The iterative G^2 family also has several possible policy variations, each characterized by a specific completion criteria and a clustering algorithm. A meta-control policy comprises, therefore, of a set of algorithms, a completion criteria, and appropriate parameters, which fully prescribe how the meta-control decision will be made. A sample policy, for example, would be to use G^2 with FFD clustering, and no more than 20 iterations. Each meta-control policy also represents a very specific balance of performance and response time, generally referred to as the policy *depth*. Response time is readily measured empirically or estimated theoretically. But how do we measure or estimate policy performance?

7.2 Measuring Performance

As already described in Chapter 6, meta-control policies making online mode selection and load-balancing decisions which attempt to maximize system gain for each given system state. We also recall that from a meta-control perspective, a *system state* is fully described by the *platform state* consisting of all processor capacities, along with the *program state* consisting of jobs, job modes, and job mode bandwidths and rewards. Policy performance for policy, π, in any given system state, s, is measured by $v_\pi(s)/v^*(s)$, where $v_\pi(s)$ is the system gain produced by π for state s, and $v^*(s)$ is the optimal system gain for s. The system state might be *fixed* for many real-time systems, but for the dynamic systems addressed by this thesis, policy performance will more typically be measured by its statistical behavior over a set or sequence of states. Let $p(s)$ denote the probability that a system will be in state $s \in S$, then the *average* performance, ψ_π, of policy π for any given set of states, S, is measured by

$$\psi_\pi = \sum_{s \in S} \frac{p(s) v_\pi(s)}{v^*(s)}. \tag{7.1}$$

But how do we evaluate π performance for larger numbers of jobs, modes and processors, when computing v^* is intractable? In Chapter 6, we replaced v^* with a $\bar{v}$ upper bound for v^*, to obtain lower bound measures of performance. These lower bound measurements might suffice for verifying compliance to certain minimal levels of performance, but they cannot serve to compute ψ_π for different π. Neither can they serve a qualitative analysis of performance, such as when attempting to evaluate whether a given family of policies, Π, is *statistically epsilon-approximate*, i.e. that for every ϵ we can find a policy $\pi \in \Pi$ such that $\psi_\pi \leq 1 - \epsilon$. For a more accurate measure of performance, we develop here a statistical model for estimating v^* for any given state. We can then evaluate ψ_π for any given policy, π, when provided with empirically-obtained v_π gains, for all $s \in S$.

7.2.1 The Statistical Model

Let Ω denote the population of all possible mode vectors in a given *application* state, s, where each mode vector, $\omega \in \Omega$, represents a unique set of selected and normalized $\{\mathsf{reward}, \mathsf{band}\}$ pairs, each $\mathsf{reward} \in [0, 1/\mathsf{njobs}]$, and each $\mathsf{band} \in [0, \beta/\mathsf{njobs}]$, where β is predetermined, and $\beta_c = \beta/\mathsf{ncpus}$

for each $c \in C$. As described in Section 6.1.1, the following three values fully characterize any given mode vector ω: the mode vector *gain*, $v = v(\omega) \in [0, 1]$, the mode vector *capacity utilization*, $u = u(\omega) \in [0, \beta]$, and the mode vector *clusterability* indicator, $\mathbf{c}(\omega) \in \{0, 1\}$. Clearly, $\mathbf{c}(\omega)$ is zero if $u(\omega) > 1$. Let $\Omega_{\mathbf{c}} \subset \Omega$ denote that portion of the population which is clusterable, then

$$\Omega_{\mathbf{c}} = \{\omega \in \Omega | \mathbf{c}(\omega) = 1\}. \tag{7.2}$$

Conversely, a statistician might prefer defining $\mathbf{c}(\omega)$ in terms of $\Omega_{\mathbf{c}}$ using the event indicator:

$$\mathbf{c}(\omega) = 1_{\{\Omega_{\mathbf{c}}\}}(\omega). \tag{7.3}$$

Let $P(\mathbf{c}|u)$ denote the clusterability probability that a randomly selected mode vector $\omega \in \Omega$ belongs to $\Omega_{\mathbf{c}}$, given that $u(\omega) = u$). Thus, $P(\mathbf{c}|u)$ is essentially equal to $\mathsf{clusterability}(\gamma^*, \mathsf{njobs}, \mathsf{ncpus}, u)$ which is computationally intractable, as described in Section 5.1.2. Nevertheless, as already discussed in Section 5.1.2, we should be able to closely approximate $P(\mathbf{c}|u)$ as a function of u and the ratio of jobs to processors.

The primary software meta-controller objective is to seek the highest gain mode vector $\omega^* \in \Omega$, which is also clusterable, i.e.

$$\omega^* = \arg \max_{\omega \in \Omega_{\mathbf{c}}} v(\omega). \tag{7.4}$$

A deterministic determination of ω^* is computationally intractable because the population size $|\Omega|$ is prohibitively large and because $\mathbf{c}(\omega)$ is also not attainable for any given ω. Our current objective, therefore, is to estimate v^* without finding ω^*.

Borrowing from terms in ordered statistics, let $\mathcal{V} = \{V_{(1)}, V_{(2)}, \cdots V_{(|\Omega|)}\}$ denote the ordered set of all $v(\omega)$ gains in the Ω population regardless of clusterability, where $|\Omega|$ represents the known size of the population. Both $V_{(1)}$ and $V_{(|\Omega|)}$ are readily determined, since they represent the lowest and highest reward vector modes available. Other $V_{(i)}$ gains are not, since they require generating all mode vector permutations and sorting them.

We now use $\mathcal{X} = \{X_{(1)}, X_{(2)}, \cdots X_{(|\Omega_{\mathbf{c}}|)}\} \subset \mathcal{V}$ to denote the ordered set of all $v(\omega)$ gains in the clusterable population $\Omega_{\mathbf{c}}$. The precise cardinality $|\Omega_{\mathbf{c}}|$ of $\mathcal{X}$, and all elements $X_{(i)} \in \mathcal{X}$ are unknown. We cannot even sample these elements directly, since testing for clusterability is NP-hard. Even if we could test sampled mode vectors for clusterability, we must take into account that the fraction of clusterable mode vectors in Ω might be exceedingly small, whereupon the required number of $\mathcal{V}$ samples for a single valid $\mathcal{X}$ sample could be prohibitive. What we want to do, therefore, is estimate $v^* = X_{(|\Omega_\mathbf{c}|)}$ as a function of $P(\mathbf{c}|u)$, $f(v)$, and $f(u|v)$, where

- $P(\mathbf{c}|u)$ represents the assumed *known* conditional probability that a $\omega \in \Omega$ belongs to the clusterable population $\Omega_\mathbf{c}$, given that its utilization is u;
- random variables $\mathbf{u}$ and $\mathbf{v}$ denote the mode vector utilization and system gain, respectively, of a randomly sampled $\omega \in \Omega$;
- $f(v)$ is the density function of the random variable $\mathbf{v}$; and
- $f(u|v)$ represents the conditional density function of $\mathbf{u}$, given that ω has a gain v.

Assuming that both $f(v)$ and $f(u|v)$ can be accurately estimated by a reasonably small random sampling of the general population Ω, then the expected optimal gain $\mathrm{E}\{v^*\}$ can be accurately determined in two phases. In the first phase, we use *total probability* to derive $P(\mathbf{c}|v)$ from $P(\mathbf{c}|u)$ and $f(u|v)$, where $P(\mathbf{c}|v)$ denotes the *clusterability* of a mode vector with gain v, i.e. the probability that a mode vector with gain v belongs to $\Omega_\mathbf{c}$. In the second phase, we use $P(\mathbf{c}|v)$, $f(v)$, and $|\Omega|$ to compute the expected optimal gain, $\mathrm{E}\{v^*\}$.

Phase 1: Computing $P(\mathbf{c}|v)$

As previously mentioned, we assume that $P(\mathbf{c}|u)$ is known, while $f(v)$ and $f(u|v)$ are estimated by a random sampling of Ω. From the *central-limit* theorem [Pap65], it follows that for a large number of jobs, both $f(u)$ and $f(v)$ will tend to a normal curve regardless of the distribution of rewards and bandwidths for each job[1], since they represent summations of several independent random variables. Even without the central-limit theorem, $f(u)$ and $f(v)$ will be normally distributed for an even smaller number of jobs when rewards and bandwidths are *uniformly* distributed. For such cases, we need a relatively small number of samples to fully characterize $f(v)$ and $f(u|v)$.

Let us take as an example a randomly generated application state for a system with 10 jobs, 10 modes per job, and a platform of 5 uniform

[1] The central-limit theorem does require, however, that all bandwidths/rewards for each job have the same probability distribution function. The other requirement, that their mean and variance exist for each job, is guaranteed by definition.

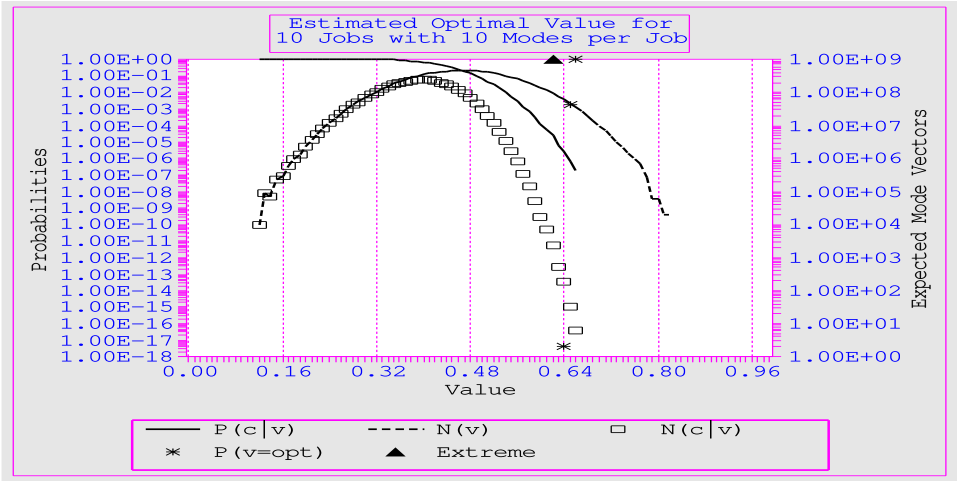

Figure 7.1: Estimating E$\{v^*\}$

processors with a total capacity of 1. Using an appropriately scaled uniform distribution over $[0, 1]$, ten reward values and ten bandwidth values were randomly generated for each job and then *monotonically paired*, i.e. the highest reward and bandwidth values were assigned to the first job mode, the second highest values were assigned to the second job mode, the third highest values were assigned to the third job mode, and so on[2]. Such a system clearly has a combinatoric population of $|\Omega| = 10^{10}$ mode vectors. The $N(v)$ plot in Figure 7.1 is equal to $|\Omega| \cdot f(v)$ as obtained from a random sampling of 10^5 mode vectors in Ω. Looking at this plot, it is not unreasonable to assume that $\mathbf{v}$ is normally distributed within the range bound by the extreme gains of $V_{(1)} = 0.12$ and $V_{(|\Omega|)} = 0.82$. As previously defined, $V_{(1)}$, here, is the gain of the lowest gain mode vector, obtained when choosing the lowest reward modes for each job, and $V_{(|\Omega|)}$ is the gain of the highest gain mode vector, obtained when choosing the highest reward modes for each job. We could then approximate $\mathbf{v} \sim N(\mu_v, \sigma_v^2)$, where μ_v is E$\{\mathbf{v}\}$ and $\sigma_v^2 = \text{Var}\{\mathbf{v}\}$. In our case, $\mu_v = 0.47$ is the mean vector gain and $\sigma_v = 0.066$ is the standard deviation.

[2]As already mentioned in Section 6.1, non-monotonic modes are of no interest to us since they cannot be part of an optimal solution.

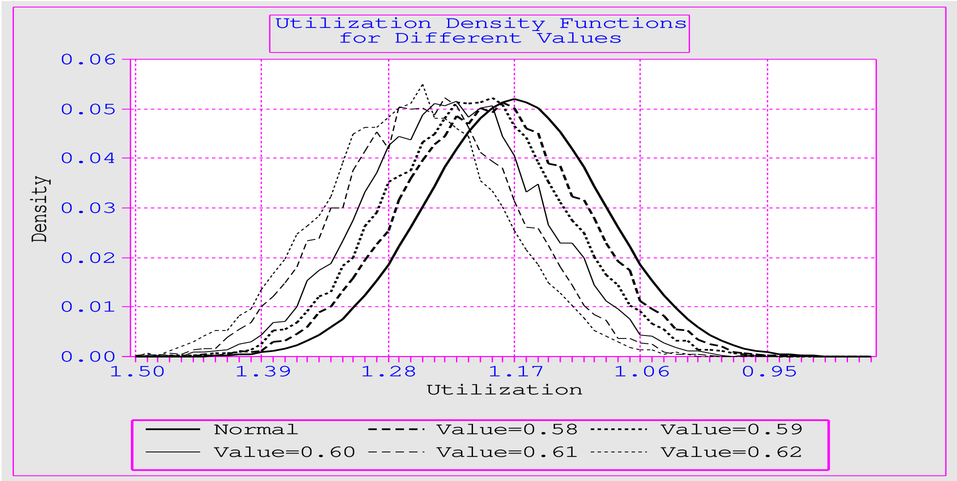

Figure 7.2: A Few Conditional Utilization Distributions

For n jobs with uniformly distributed bandwidths and rewards, and known extreme gains $V_{(1)}$ and $V_{(|\Omega|)}$, we can estimate the mean ($\hat{\mu}_v$) and the variance ($\hat{\sigma}_v^2$) for $f(v)$ without even sampling, by[3]

$$\hat{\mu}_v = \frac{V_{(|\Omega|)} + V_{(1)}}{2}, \tag{7.5}$$

and

$$\hat{\sigma}_v^2 = \frac{(V_{(|\Omega|)} - V_{(1)})^2}{12n}. \tag{7.6}$$

These normal approximations are not necessarily accurate for small numbers of jobs, n, and we will therefore avoid using them when estimating $\mathrm{E}\{v^*\}$. Nevertheless, they provide us with a good framework for qualitative reasoning about $\mathrm{E}\{v^*\}$, as we do later.

Figure 7.2 shows us a few examples of the conditional density function, $f(u|v)$, for several v gains which are of particular interest, as we later see. All $f(u|v)$ densities in our example look like normal distributions with very similar variances and with a mean utilization which increases in a linear

[3]Each job j bandwidth, $b_j \sim U(0, \frac{1}{B})$; $S_n = \sum_{j=1}^{n} b_j$; by the Central Limit Theorem [BF81], $S_n \sim N(n \cdot \mathrm{E}\{b_j\}, n \cdot \mathrm{Var}\{b_j\})$ where $\mathrm{E}\{b_j\} = \frac{1}{2B}$, $\mathrm{Var}\{b_j\} = \frac{1}{12B^2}$.

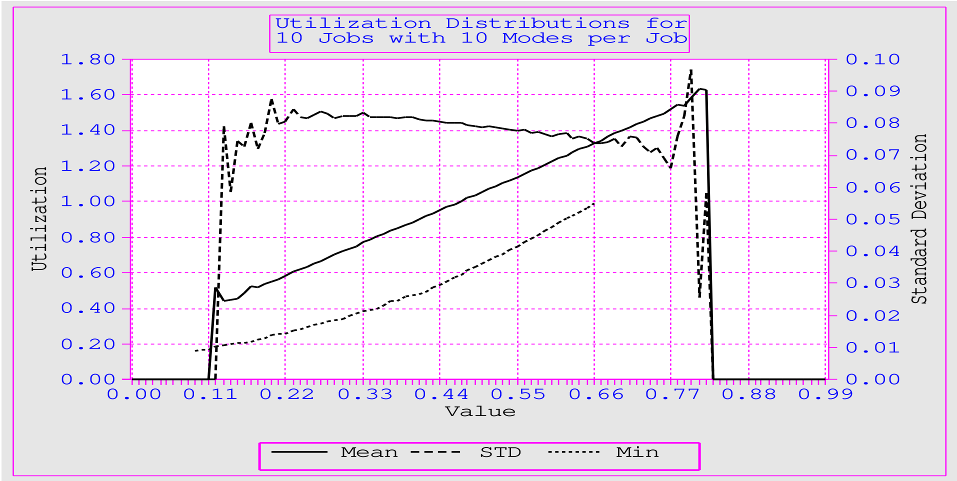

Figure 7.3: Typical Utilization Distributions

fashion for larger v gains. Thus, we can approximate the conditional random variable $\mathbf{u}$ given $\mathbf{v} = v$ by

$$\mathbf{u}|v \sim N(\mu(v), \sigma(v)^2), \tag{7.7}$$

where sampled mean $\mu(v)$ values and $\sigma(v)$ standard variations for 10^5 samples are plotted in Figure 7.3. We see that $\mu(v)$ is predominantly linear over v while $\sigma(v)$ is relatively constant within much of the range bounded by $V_{(1)}$ and $V_{(|\Omega|)}$. Sharp variations in $\sigma(v)$ occur only for extreme v values where the number of mode vector samples is very small. The Min graph in Figure 7.3 indicates the minimal utilization $U_{min}(v)$ in Ω for each gain v, as ϵ-approximated by dynamic programming[4]. Mode vectors with utilizations which exceed 1.0 are not clusterable, so that $U_{min}(v^*) < 1.0$. All v with $U_{min}(v) \geq 1.0$ are not candidates for v^*. In our particular example, $U_{min}(0.66) = 1.0$, so that we know that $v^* \leq 0.66$. Thus, 0.66 serves as an upper bound for v^* denoted by $\bar{v}$, i.e.

$$\bar{v} = \arg\max_{v}\{v | U_{min}(v) < 1.0\}. \tag{7.8}$$

[4]These ϵ-approximate $U_{min}(v)$ values are a by-product of the QDP dynamic programming phase.

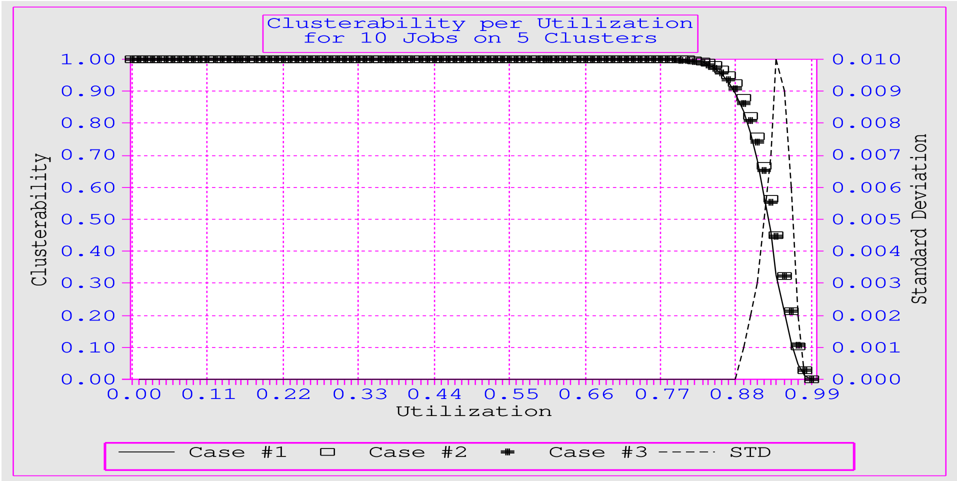

Figure 7.4: Utilization Clusterablity

We can see, therefore, that $\mu(v)$ and $\sigma(v)$ exhibit very predictable behavior in the range of candidate v gains for our example. We also note that

$$\mu(v) - U_{min}(v) > 4 \cdot \sigma(v), \tag{7.9}$$

so that we would need much more than 10^5 samples per v to pick up a mode vector sample with $u = U_{min}(v)$. The lowest utilization mode clusters are those which are of the greatest interest when seeking v^*. Thus, the assumption of a normal distribution for $\mathbf{u}|v$ is quite essential for the model.

As already mentioned, we can assume that the utilization clusterability $P(\mathbf{c}|u)$ is already known because it can be determined independent of the current application state, knowing only the number of jobs and processors. Figure 7.4 illustrates this by plotting utilization clusterability probabilities for 3 randomly generated applications which are identical only in the number of jobs (10) and processors (5). Each of the Case graphs are histograms which provide the number of mode vectors found to be clusterable in any given utilization interval. As indicated by the standard deviation graph, we can see that differences in the clusterability of all utilizations are negligible.

Armed with $f(v)$, $P(u|v)$, and $P(\mathbf{c}|u)$, we can then compute $P(\mathbf{c}|v)$ using total probability [Pap65] by

$$P(\mathbf{c}|v) = \begin{cases} v > \bar{v} & 0 \\ v \leq \bar{v} & \int_{-\infty}^{\infty} P(\mathbf{c}|u) \cdot f(u|v)du \end{cases} . \tag{7.10}$$

This $P(\mathbf{c}|v)$ result is plotted in Figure 7.1, where gain clusterability is shown to be 1.0 for all gains below $v \leq 0.4$, since the bulk of mode vector utilizations for these gains are in a range where $P(\mathbf{c}|u) = 1.0$. The $P(\mathbf{c}|v)$ plot dips down to 10^{-7} as v approaches $\bar{v} = 0.66$ where nearly all mode vectors have non-clusterable utilizations.

Phase 2: Estimating v^*

The computation of $\mathrm{E}\{v^*\}$ from $P(\mathbf{c}|v)$, $f(v)$, and $|\Omega|$, is relatively straightforward when using piecewise approximation over a discrete gain domain. The gain domain of $v \in [0,1]$ is broken up into K equal intervals: $\{I_k\}, k = 1, 2, \cdots, K$, where $I_k = (\frac{k-1}{K}, \frac{k}{K}]$. Let $n(k)$ denote the number of population mode vectors, ω, such that $v(\omega) \in I_k$. Rather than unnecessarily assume a normal distribution for $\mathbf{v}$, we prefer to randomly test N samples from within the population to obtain a non-biased estimator $\hat{n}(k)$ for $n(k)$

$$\hat{n}(k) = \frac{n_k \cdot |\Omega|}{N}, \tag{7.11}$$

where n_k is equal to the number of sampled gains belonging to interval I_k. The standard error s_n of such an estimation is then computed by [BF81]

$$s_n = \frac{\sqrt{\hat{n}(k)(1 - \hat{n}(k)/N)}}{N}. \tag{7.12}$$

Using inferal statistics and the cumulative normal distribution function Φ, we can then assume, with a 0.999 confidence level, that the following $\bar{n}(k)$ is indeed an upper bound for the actual number of population mode vectors, $n(k)$, in I_k:

$$\bar{n}(k) = \hat{n}(k) + 3.291\sqrt{\hat{n}(k)[1 - \hat{n}(k)/N]} \geq n(k), \tag{7.13}$$

since $\Phi(3.291) = 0.999$. Let $Q(k)$ denote the probability that *none* of the $n(k)$ mode vectors in interval I_k is clusterable, and let $\bar{p}(k) = P(\mathbf{c}|v = \frac{k}{K}) \geq P(\mathbf{c}|v \in I_k)$ denote an upper bound for the probability that any given mode vector in interval I_k is clusterable. Then with great confidence we can determine that $Q(k)$ is certainly greater than the probability that none of the $\bar{n}(k) \geq n(k)$ mode vectors is clusterable when assuming an optimistic measure of clusterability, i.e.

$$Q(k) \geq [1 - \bar{p}(k)]^{\bar{n}(k)}. \tag{7.14}$$

Seeking the highest interval I_k with a clusterable mode vector, this can be modeled as a *Bernoulli* trial with no more than K independent[5] experiments. Each experiment $t = 0, 1, \cdots, K-1$ succeeds if a clusterable mode vector is found in interval I_{K-t}. The probability of success in experiment t is clearly $1 - Q(K-t)$. The Bernoulli trial ends with the first successful experiment. Letting random variable $\mathbf{t}$ represent the number of experiments in a random trial, then

$$P(v^* \in I_{K-t}) = P(\mathbf{t} = t) = (1 - Q(K-t)) \cdot \prod_{i=0}^{t-1} Q(K-i), \tag{7.15}$$

since all experiments prior to t must first fail before experiment t succeeds. $\mathrm{E}\{v^*\}$ is then estimated by

$$\lim_{K \to \infty} \mathrm{E}\{v^*\} = \sum_{t=0}^{K-1} \frac{K-t}{K} \cdot P(\mathbf{t} = t). \tag{7.16}$$

7.2.2 A Concrete Example

Plot $N(\Omega_{\mathbf{c}}|v)$ in Figure 5.5 provides the expected number of clusterable mode vectors in the I_k interval that v belongs to, i.e.

$$N(\Omega_{\mathbf{c}}|v) = \mathrm{E}\{n(\lceil v/K \rceil)\}. \tag{7.17}$$

In this case we used $K = 100$. Here we can see that the expected number of clusterable mode vectors for a population of $|\Omega| = 10^{10}$ is close to $2 \cdot 10^8$ for the interval I_{40} which begins at $v = 0.40$, and that it tapers down to 7 mode vectors in interval I_{66} which begins at $v = 0.66$. Graph $P(v = opt)$ plots the probability that v^* is in the interval to which v belongs. For $v \in I_k$, this probability is essentially equal to $P(\mathbf{t} = K - k)$ in Formula (7.15). As implied by graph, $P(v = opt)$, the probability that $v^* \in I_{66}$, is nearly 1.0, so that $\mathrm{E}\{v^*\}$ is somewhere in between 0.66 and 0.67. We can obtain more

[5] It has already been established that $\hat{p}_k$ values have Poisson distributions which can be treated as independent [Pap65].

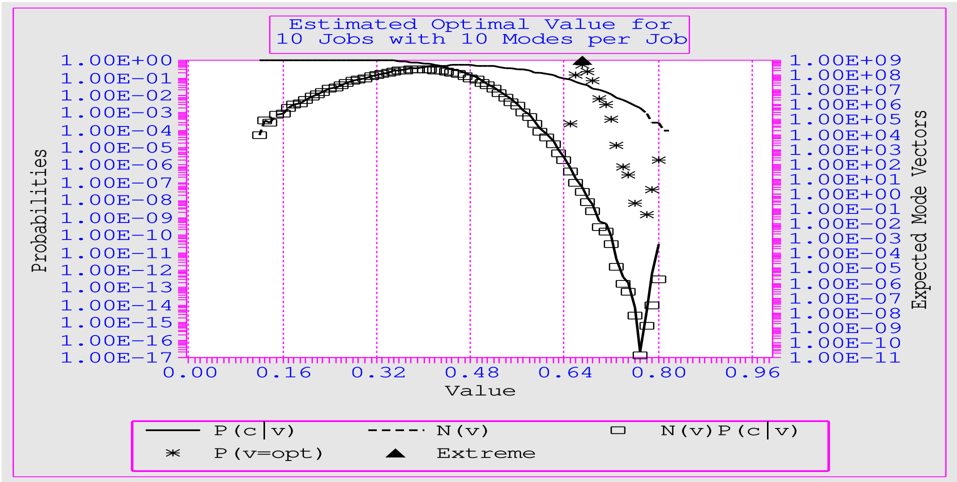

Figure 7.5: The Effects of Spurious $\sigma(v_k)$ on $\mathrm{E}\{v^*\}$

accurate estimations of $\mathrm{E}\{v^*\}$ by breaking up the gain range into a larger number K of intervals. Also noted in Figure 7.1 is the Extreme gain actually found within the 10^5 random samples of Section 7.2.1. The clusterability of each sample was tested here using FFD which very rarely fails to establish clusterability when it exists (Section 5.2). This extreme gain was found in interval I_{62} which begins at $v = 0.62$. This is significantly less than $\mathrm{E}\{v^*\}$ because $P(\mathbf{c}|v)$ is lower than 10^{-5} for larger v gains. This demonstrates the severe limitations of attempting to sample $\mathcal{X}$ directly.

Figure 7.5 shows us what happens if we don't set $P(\mathbf{c}|v) = 0$ for all $v > \bar{v}$ in Formula 7.10. Here we see how even a slight imprecision in $\sigma(v)$ can effect the outcome of $\mathrm{E}\{v^*\}$. Less significant is the way $P(\mathbf{c}|v)$ suddenly rises for $v > 0.80$ due to the $\sigma(v)$ spikes shown in Figure 7.3 for those gains. We recall that these spikes are a result of the very small number of mode vectors sampled in those intervals. These introduce negligible expected fractions of clusterable mode vectors which reach 10^{-3} for $v = 0.80$. More significant are the very marginal $\sigma(v)$ ripples at $0.66 < v < 0.70$, which introduce expected numbers of clusterable mode vectors, which cannot be disregarded, and which ultimately raise $\mathrm{E}\{v^*\}$ beyond the deterministically determined $\bar{v}$ upper bound. These $\sigma(v)$ ripples are a consequence of a small number of samples vectors for those intervals. This is best overcome by smoothing the $\sigma(v)$ curve or by setting $P(\mathbf{c}|v)$ to zero for $v > \bar{v}$. The latter approach is simpler, but it does not adequately address less loaded application states where μ_u and $\bar{v}$ would be high enough to contain less stable $\sigma(v)$ approximations as well.

v^* Proximity to $\bar{v}$

Using the above example, we make the following observations regarding the proximity of v^* to $\bar{v}$. Is $\bar{v}$ a tight or loose upper bound for v^*? As illustrated by Figure 7.4, $\mathrm{E}\{v^*\}$ will essentially be equal to $\bar{v}$ for $N(\Omega_{\mathbf{c}}|\bar{v}) >> 1$. By definition, $N(\Omega_{\mathbf{c}}|\bar{v})$ is always non-zero and positive. For smaller $N(\Omega_{\mathbf{c}}|\bar{v})$, $\mathrm{E}\{v^*\}$ is located somewhere between $\bar{v}$ and the highest v, which fulfills the requirement that $N(\Omega_{\mathbf{c}}|v) >> 1$. The actual proximity of $\mathrm{E}\{v^*\}$ to $\bar{v}$ is therefore determined by the nature of $N(\Omega_{\mathbf{c}}|v)$ as it approaches $\bar{v}$. The greater the decline, the closer the proximity. In the example of Figure 7.1, $N(\Omega_{\mathbf{c}}|v)$ has a high peak with a very sharp slope, whereupon $\mathrm{E}\{v^*\}$ is readily determined to be where $N(\Omega_{\mathbf{c}}|v)$ drops below 1. A smaller work load, i.e. an application with lower utilizations, would not effect $N(v)$, but it would reduce the average utilization and subsequently move $P(\mathbf{c}|v)$ to the right. This will cause $N(\Omega_{\mathbf{c}}|v)$ to rise even higher and then decline even faster, so that once again, $\mathrm{E}\{v^*\}$ must converge to $\bar{v}$. In a larger work load, on the other hand, $P(\mathbf{c}|v)$ would move to the left, causing $N(\Omega_{\mathbf{c}}|v)$ to decline much earlier and to possibly even twist as it declines. A lower peak and a more gradual slope where $N(\Omega_{\mathbf{c}}|v)$ dips below 1 would cause $\mathrm{E}\{v^*\}$ to be somewhat distanced from $\bar{v}$. A significantly smaller mode vector population would have a similar effect since the $N(\Omega_{\mathbf{c}}|v)$ peak will be much reduced with a less pronounced downward slope.

We therefore conclude that $\bar{v}$ is a very tight upper-bound for v^* when dealing with very large populations with maximum utilizations that do not exceed double the total capacity. This result is actually quite intuitive, since the clusterability of high-utilization mode vectors is clearly higher when clustering larger numbers of smaller items.

7.2.3 Suggested Model Improvements

Our current statistical model for approximating v^* can be improved in several ways. As we have pointed out in Section 7.2.2, there are difficulties in estimating $f(u|v)$ for v gains which have very small populations. This can be improved upon by sampling the population in a way which will bias extreme gain mode vectors, i.e. by using random mode selection distributions which are biased to the lowest and highest job modes. This sampling cannot be used to estimate $f(v)$, so we have to sample Ω once again, selecting job modes with a uniform distribution to estimate $f(v)$. We can also use smoothing techniques to remove variance spikes in gain intervals for which very few mode vectors were sampled.

Many other aspects of our experimentation leave much to be desired. We would like to test for higher K values, varying maximum utilizations, and a wider range of population sizes. We would like to properly evaluate biases introduced by FFD-estimation of $P(\mathbf{c}|u)$ and to devise appropriate bias-correction techniques. Nonetheless, we contend that our results already provide ample evidence to support the validity of our statistical model and subsequent conclusions.

7.3 Estimating QDP Performance

We now extend our statistical model to estimate QDP performance for a given state as a function of the bandwidth quantum, q. As described in Section 6.2, QDP uses a combination of bandwidth quantization, dynamic programming, and conventional binpacking techniques to provide v_π gains which we will show to be statistically ϵ-approximate for v^* as a function of q. The quantum, q, determines the maximum number of dynamic programming states, so that QDP time complexity is linearly proportional to $1/q$. By establishing QDP cost and performance as a function q, we can determine which q will best suit a given application in a given state. Such an analysis would be carried out offline for typical system states.

The QDP Strategy

As described in Section 6.2, the QDP algorithm produces a mode vector ω_π with a system gain $v_\pi = v(\omega_\pi)$ in two distinct phases. In the first phase, QDP uses *dynamic programming* to simultaneously produce an subset Ω_π of mode vectors. Mode vectors in Ω_π are FFD tested for clusterability in the second phase, whereupon the most valuable clusterable mode vector is adopted as ω_π. Prior to dynamic programming, QDP uses bandwidth *quantization*, i.e., all bandwidths are rounded up to the nearest multiple of a given *quantum*, q, to ensure that $N = |\Omega_\pi|$ will not exceed a given size. Higher QDP gains can be traded off for shorter response times by controlling

q in the first phase and by deciding which mode vectors should be tested for clusterability in the second phase. The dynamic programming phase is designed to ensure that each mode vector in Ω_π closely approximates the highest available mode vector gain, v, for a specific maximum utilization, u. Letting $\omega_{(n)}$ denote the mode vector with the n-th highest gain in Ω_π, then

$$v(\omega_{(n)}) \simeq \max_{\omega \in \Omega} \{v(\omega) | u(\omega) \leq \frac{n}{N}\}. \tag{7.18}$$

As in Section 7.2.1, we break up the gain domain of $v \in [0, 1]$ into K uniform intervals: $\{I_k\}, k = 1, 2, \cdots, K$, where $I_k = (\frac{k-1}{K}, \frac{k}{K}]$. The primary QDP objective is to produce for each I_k interval a set of mode vectors with the lowest utilization u available, i.e. with the highest potential for clusterability.

Statistically ϵ-Approximate Behavior

A meta-control policy π is considered *statistically* ϵ-approximate for any given state if the *expected* gain v_π produced by π is sufficiently close to the expected optimal gain v^*, i.e.

$$\mathrm{E}\{v^*\} - \mathrm{E}\{v_\pi\} \leq \epsilon \cdot \mathrm{E}\{v^*\}. \tag{7.19}$$

As described in Section 7.2.1, Formula (7.15) provides the probability $P(v^* \in I_k)$ that v^* is in any given interval I_k. To become statistically ϵ-approximate, we require that the probability that v_π will be found in the same interval, $P(v_\pi \in I_k)$, should be sufficiently close to $P(v^* \in I_k)$, for all k with $P(v^* \in I_k) > 0$. Lacking ϵ-approximate binpacking alternatives, there is very little hope for a *deterministic* ϵ-approximate solution to our problem. This *statistical* measure of performance is more than adequate when optimizing for *average* performance, as in software meta-control for a sequence of dynamic states.

Comparing $P(v_\pi \in I_k)$ and $P(v^* \in I_k)$

Let Ω^k denote the subset of Ω mode vectors which have gains in I_k, i.e.,

$$\Omega^k = \{\omega \in \Omega | v(\omega) \in I_k\}. \tag{7.20}$$

In a similar fashion, we use Ω_π^k to denote the intersection of Ω^k and Ω_π, i.e.,

$$\Omega_\pi^k = \Omega^k \cap \Omega_\pi. \tag{7.21}$$

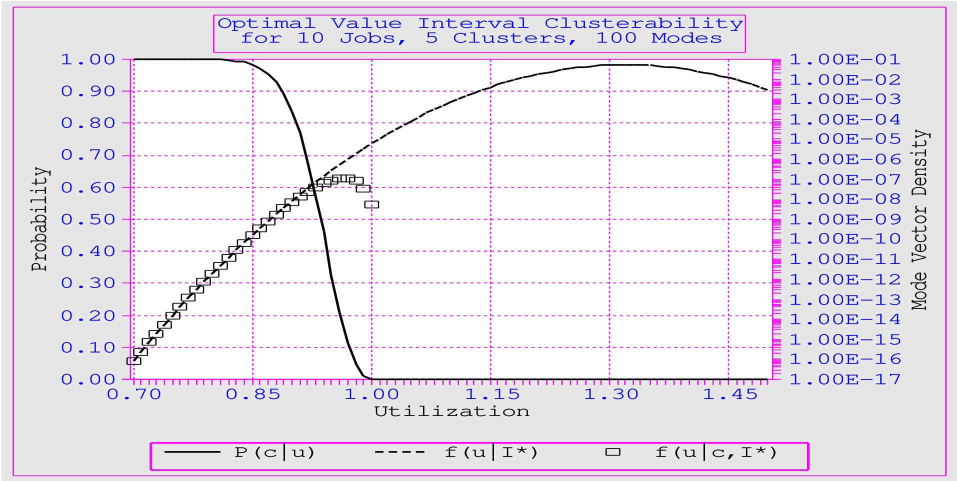

Figure 7.6: Clusterability within a Given Gain Interval

To obtain an ϵ-approximate behavior, we need to ensure that Ω_π^k can adequately represent Ω^k, i.e., that the probability of finding a clusterable mode vector in Ω_π^k will be ϵ-approximate to the probability of finding a clusterable mode vector in Ω^k.

The probability that no $\omega \in \Omega^k$ will cluster is determined by $Q(k)$ in Formula (7.14), and $P(v^* \in I_k)$ of Formula (7.15) is the primary factor in $Q(k)$. In a similar fashion, we obtain

$$P(v_\pi \in I_k) = (1 - Q_\pi(k)) \cdot \prod_{i=k+1}^{K} Q_\pi(k), \tag{7.22}$$

where $Q_\pi(k)$ denotes the probability that no $\omega \in \Omega_\pi^k$ will cluster.

Figure 7.6 illustrates the basic difference between $Q(k)$ and $Q_\pi(k)$ in reference to the example of Section 7.2.2. Let I^* denote interval I_{66}, which was the only one with $P(v^* \in I_k) > 0$. The graph of $P(\mathbf{c}|u)$ plots the known utilization clusterability which is assumed to be interval independent, and the graph of $f(u|I^*)$ plots the density of mode vectors in Ω^{66} as a function of u. The graph of $f(u|C, I^*)$ plots the density of *clusterable* mode vectors in Ω^{66} and it must, of course, drop to zero for $u > 1.0$. We can see here that the bulk of clusterable mode vectors in Ω^k is just below $u = 1.0$ and that it

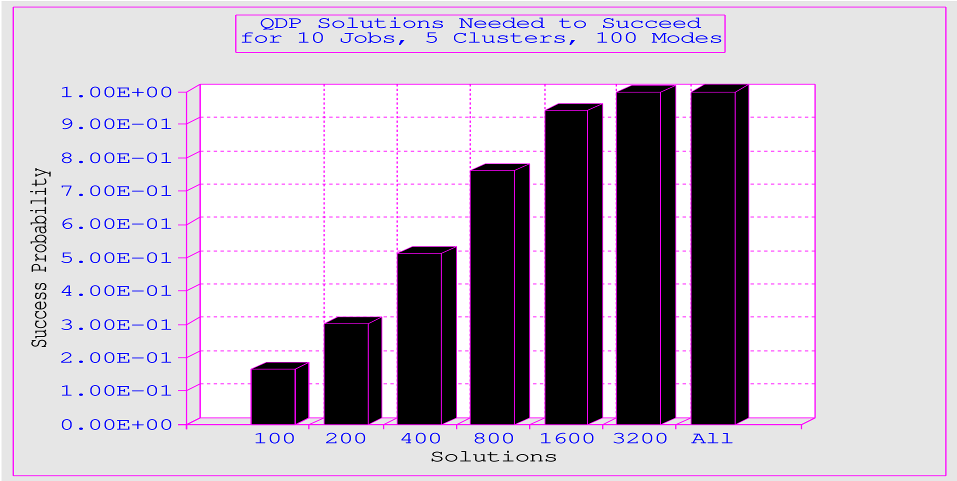

Figure 7.7: Probability $P(v_\pi \in I^*)$ as a Function of $|\Omega_\pi|$ for $K = 100$

is but a small fraction of the mode vectors in Ω^{66}. All QDP mode vectors in Ω_π^k have the minimal u available. Referring to Formula 7.18, we are even guaranteed $u(\omega_{(n)}) \leq n/N$, i.e. for each $\omega \in \Omega_\pi$, we know that its utilization will never exceed a given value. In Section 7.2.1, we used $U_{min}(v)$ to denote the minimal utilization available with system gain v which is attainable by dynamic programming. In a similar fashion, we now use $U_{min}(I_k)$ to denote the minimal utilization available with a gain in interval I_k. QDP provides $|\Omega_\pi^k|$ mode vectors in interval I_k with utilizations which can be assumed to be less than $U_{min}(I_{k+1})$, by virtue of Formula (7.18). Using p_k^π to denote the worst-case probability that a mode vector in Ω_π^k will cluster, then

$$p_k^\pi = P(\mathbf{c}|U_{min}(I_{k+1})). \tag{7.23}$$

We can now compute the worst-case probability $Q_\pi(k)$ that *no* mode vectors in Ω_π^k will cluster as

$$Q_\pi(k) \geq (1 - p_k^\pi)^{|\Omega_\pi{}^k|}. \tag{7.24}$$

An ϵ-Approximate q

Referring again to the example of Section 7.2.2 in which $K = 100$, Figure 7.7 uses Equation (7.22) to provides the probability that v_π will fall into the same interval as v^* for different sizes of $|\Omega_\pi|$. We see here that a very small number of 100 uniformly distributed mode vectors in Ω_π will provide only a 16 percent guarantee that v_π will fall in the interval $I^* = I_{66}$. With 3200 uniformly distributed mode vectors, we already have a 99 percent degree of certainty that v_π will fall inside I^*.

For an ϵ-approximate v_π, we must carry out our analysis for $K = 1/\epsilon$. We use QDP with a relatively coarse q to estimate $U_{max}(k)$, thereby obtaining p_k^π via Formula (7.23). Our goal, now is, to determine a minimal $|{\Omega_\pi}^k|$ which will bring the $Q_\pi(k)$ of Formula (7.24) close enough to the $Q(k)$ of Formula (7.14) so that $P(v_\pi \in I_k) = P(v^* \in I_k)$. This minimal $|{\Omega_\pi}^k|$ can be derived iteratively, whereupon it dictates the overall size of Ω_π, $N = K \cdot |{\Omega_\pi}^k|$. The ϵ-approximate q is then computed by

$$q = \frac{\mathsf{ncpus} \cdot u_{max}}{\mathsf{njobs} \cdot \mathbb{N}}, \tag{7.25}$$

where u_{max} is the largest possible mode vector utilization.

A Qualitative Analysis

In the above example, only one interval was relevant, so all we need is a $Q_\pi(k)$ of zero to guarantee that v_π will be in the same interval. We also note that an $|\Omega_\pi|$ of 3200 is only a fraction of an $|\Omega|$ of 10^{10}, thus illustrating how effective our QDP strategy is in producing Ω_π mode vectors with the highest clusterability potential per gain. QDP dynamic programming can be used to provide us with any desirable number of low-utilization mode vectors in any given gain interval. We are primarily interested in the gain intervals just below $\bar{v}$, which is equal to 0.66 in our example. As illustrated in Figure 7.8, their low utilization make them more clusterable than randomly selected mode vectors in that vicinity, by several orders of magnitude.

We only need *one* clusterable mode vector for any given gain interval, so what counts is the probabilities $Q(k)$ and $Q_\pi(k)$ that *none* of the mode vectors in a given set will be found clusterable. These probabilities are clearly a function of clusterability of the mode vectors in the set and the size of the set. When dealing with Ω, clusterability is significantly lower because of the higher utilizations in the set. We can therefore obtain similar levels of $Q_\pi(k)$ with a much smaller set of low utilization mode vectors. To

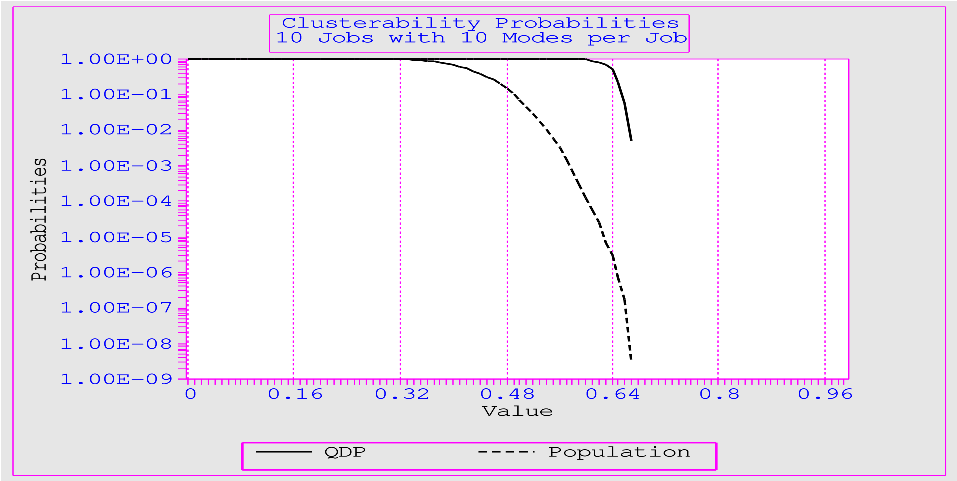

Figure 7.8: Gain Clusterability for QDP Solutions

cater to a smaller ϵ we require a larger number K of narrower gain intervals. We can always raise $|\Omega_\pi^k|$ to the required levels by choosing an appropriately small quantum q. We have yet to determine how doubling K will effect the q necessary to maintain $Q_\pi(k) \simeq Q(k)$. To be considered *fully* ϵ-approximate, we have to show that $O(1/q) \leq O(K)$.

Empirical Verification

We now wish to verify the validity of our statistical model by extensive Monte Carlo experimentation with our current version of QDP. Table 7.1 provides results for 13 batches of experiments with 40 randomly-generated experiments in each batch. Each batch has a different number of jobs, processors, and modes per jobs. Each experiment consisted of a randomly-generated system state for the given number of jobs, processors, and modes. We use $\hat{\Psi}_\pi$ to denote the *estimated* average performance $\mathrm{E}\{v_\pi/v^*\}$ for a random sequence of system states[6]. We use Ψ_π to denote the actual aver-

[6] We have yet to verify analytically that a sequence of 40 states is sufficient, but we have found that similar results are obtained when testing for additional state sequences with the same number of states.

batch	jobs	cpus	modes	q1	STD	samples	$\bar{\Psi}_\pi$	$\hat{\Psi}_\pi$	Ψ_π
1	5	3	15	0.025	0.064	200	0.927	0.986	0.989
2	6	3	18	0.025	0.053	600	0.931	0.972	0.993
3	6	4	24	0.025	0.039	2000	0.948	0.992	0.997
4	6	4	36	0.017	0.033	2000	0.936	0.989	
5	7	3	21	0.025	0.075	2000	0.929	0.953	0.990
6	10	5	40	0.031	0.031	20000	0.941	0.946	
7	10	5	60	0.021	0.022	20000	0.934	0.929	
8	10	5	60	0.021	0.025	40000	0.933	0.939	
9	10	5	60	0.021	0.025	80000	0.934	0.933	
10	11	5	44	0.031	0.025	20000	0.954	0.949	
11	11	5	66	0.021	0.024	20000	0.938	0.929	
12	12	5	48	0.031	0.017	20000	0.955	0.958	
13	12	5	72	0.021	0.024	20000	0.939	0.936	

Table 7.1: QDP Performance

age performance as obtained by actually exploring every combinatoric mode vector to determine v^*. Ψ_π measurements can be obtained only for batches with small numbers of jobs, modes, and processors. An additional $\bar{\Psi}_\pi$ performance measure is a lower bound for Ψ_π which is obtained by assuming $\bar{v}$ as the performance scale for each experiment. The $\bar{v}$ for each experiment was provided by QDP with a very fine bandwidth quantum q. Such a $\bar{v}$ is an ϵ-approximation of v^* for a single cluster, and it therefore represents the highest available gain with a utilization that does not exceed 1.0.

Our current version of QDP was carried out several times for each experiment with decreasing q values until v_π reached saturation, i.e. until v_π ceases to increase. This final q value is therefore referred to, here, as the *saturation* quantum, $q1$, since we naturally assumed that v_π had converged to v^*. Batch 10, for example, included the experiment described in Section 7.2.2. We see here that the 40 sample experiments provided us with an estimated average performance of $\hat{\Psi}_\pi = 0.949$ with a standard deviation of 0.025. The QDP-provided lower bound of $\bar{\Psi}_\pi = 0.954$ is well within the margin of error so that we can reasonably infer with a high confidence level that Ψ_π is indeed very close to $\bar{\Psi}_\pi$. This appears to be true for all the large population batches 6 through 13. This sharply contrasts with the results of batches 1 through 4, where $\hat{\Psi}_\pi$ is much higher than $\bar{\Psi}_\pi$. All of these results match our predictions in Section 7.2.2, since the populations in batches 1 through 4 are very small while that of batch 10 is very large. Indeed, batches 1 through 3 had small enough populations to enable the determination of Ψ_π, finding it to correlate very well with our estimated $\hat{\Psi}_\pi$. Batch 5 might appear to be an exception, but its large standard deviation of 0.075 puts $\hat{\Psi}_\pi = 0.953$ well within the range of $\Psi_\pi = 0.99$ with a reasonable level of confidence. Thus we have verified that $\bar{\Psi}_\pi$ is very pessimistic for smaller Ω populations, but that this pessimism shrinks as the population grows.

Batches 5 through 7 were designed to test the adequacy of the sampled population for making our assessments. We can see that quadrupling the sampled population did not significantly alter our results, so that the sampling size of 20000 appears to be adequate.

At first glance, these experiments appear to disprove our contention that QDP is statistically ϵ-approximate, at least when dealing with larger Ω populations. This is not so, however, since the saturation quantum, $q1$, for these batches have clearly not reached the levels required by Figure 7.7. Using Equation (7.25), we can deduce that a quantum, $q1 = 0.025$, will provide us only with $|\Omega_\pi| \simeq 400$ in batch 10. Referring again to Figure 7.7, this gives us at most a 50 percent chance that v_π will share the same interval as v^*. Why, then, did v_π saturate so early? A deeper examination revealed that our current QDP version was generating duplicate mode vectors for adjacent dynamic programming states, so that the size of Ω_π was simply not growing. Even $q1$ saturation quanta were dominated by duplicates so that $|\Omega_\pi|$ in batch 10 was actually much less than 400. Saturation was reached only because the rate of duplication caught up with the growth rate of q so that $|\Omega_\pi|$ could not effectively grow beyond a certain point. Thus even these results have served to verify our model for estimating QDP performance when considering the number of *unique* mode vectors produced by QDP in each experiment. An important by-product of these experiments was that we came to realize that we have a serious mode vector duplication problem in our current QDP version which was seriously hampering our ability to realize the ϵ-approximate potential. Possible solutions to this problem are suggested in Section 7.3.1. These also suggest how QDP time complexity might be be reduced to be a sublinear function of $1/q$, so that finer q values are more affordable.

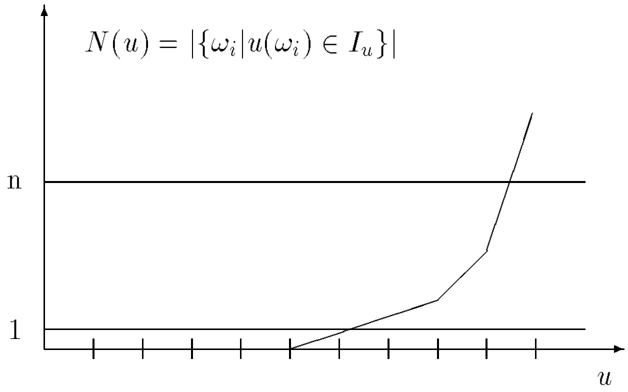

Figure 7.9: Desired Ω_π Utilization Distribution

7.3.1 Realizing QDP Potential

We have shown here how the optimal system gain v^* can be approximated for any given system state to serve as an absolute performance scale for v_π gains produced by any given software meta-control policy. We have also shown how the QDP bandwidth quantum, q, can be chosen to ensure that expected v_π results will ϵ-approximate v^*. Many aspects of our statistical model have been verified by extensive Monte Carlo experimentation. These experiments have led us to conclude that our current QDP implementation must be revised to realize its ϵ-approximate potential.

QDP revisions should be carried out in two domains. In the first domain, we must seek methods for maximizing the diversity of the solutions produced without sacrificing their ability to provide maximum gain for minimum utilization. The point here is to avoid duplicate mode vectors which effectively reduce the size of Ω_π^k. In the second domain, we would like to reduce the QDP time complexity to be a sublinear function of $1/q$, so that finer q values are more affordable. Our motto here is that there is no point in generating QDP solutions in Ω_π that will not be used. To increase QDP diversity, we suggest introducing stochastic elements into the dynamic programming phase. What we propose here is that when determining the job mode which will maximize system gain for each dynamic programming state, we use a randomized score which is strongly biased to higher system gains as well as higher levels of fragmentation.

To reduce QDP time complexity to a sublinear function of $1/q$, we propose integrating the Ibarra and Kim scheme (Section 6.6) into our QDP framework in a manner that will provide us with more than one mode vector per dynamic programming state. As illustrated in Figure 7.9, the number of mode vectors per dynamic programming state should be designed in such a way that it is substantially high for higher utilizations, and subsequent higher gains, while dropping exponentially to one mode vector only for the dynamic programming state that corresponds to $u = \bar{u}$. The number of mode vectors for lower utilizations should be zero. This scheme will guarantee that there will always be a clusterable mode vector in Ω_π while maximizing the probability that the highest clusterable gain will be equal to v^*. This regulated distribution of mode vectors will enable us to increase the density of mode vectors in the higher gain intervals without a proportionate increase in the total number of mode vectors by doing so at the expense of lower gain intervals where $P(v^* \in I_k) = 0$. Incorporating the Ibarra and Kim scheme is relatively straightforward, but we have yet to determine the precise nature of the distribution function which should be used to determine the number of mode vector solutions which should be retained for each dynamic programming state. We note, for example, that the density of Ω_π mode vectors in any gain interval cannot exceed the density of Ω, so that the desirable Ω_π distribution might actually rise as u decreases before it begins to drop exponentially.

We have yet to verify that these improvements will indeed help to realize the ϵ-approximate potential that QDP appears to have. We would also like to more carefully analyze QDP duplication behavior in our current QDP version.

7.4 Optimal Policy-Making

Let Π represent the set of relevant meta-control policies, and λ a policy-making policy for determining which meta-control policy $\pi \in \Pi$ to activate for each system state transition. We define a *stationary* λ policy to be one that always chooses the same $\pi \in \Pi$ for all decision epochs. An *adaptive* λ policy, on the other hand, may alter its choice of π with time. Given the set of anticipated system states, S, then an optimal stationary policy-maker, λ^*, could choose an optimal meta-control policy, $\pi^* \in \Pi$, if it knows how much time will be spent within each state $s \in S$. If the set of system states, S, is substantially well-defined and readily identified, then an *adaptive* λ

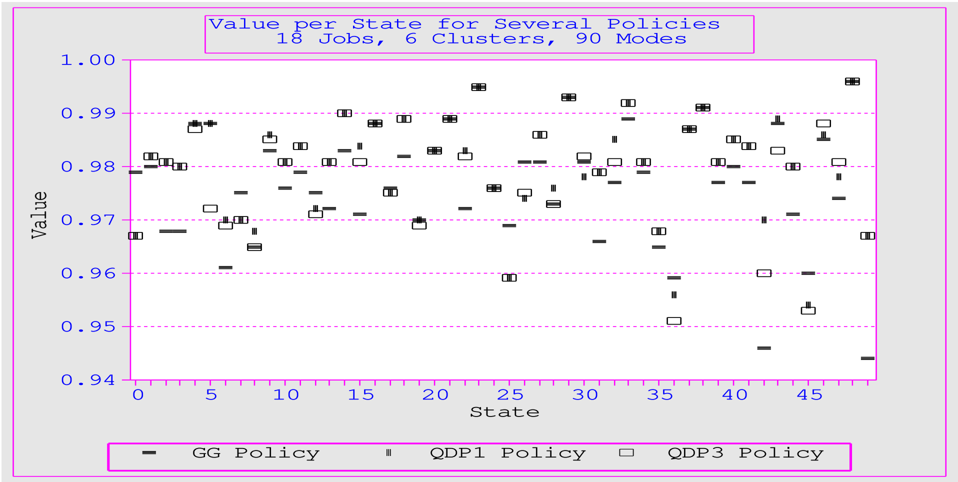

Figure 7.10: Typical Gain Distributions for Sample Policies

policy, might determine an optimal $\pi^*(s)$ meta-control policy for each state, $s \in S$.

The number of system states, however, is often enormous, whereupon it may be impractical to select π as a function of the current state. A system state is defined by the current real values for all bandwidths, rewards, and capacities in the system. *Matching* states are states which warrant equivalent optimal π policies, so that and it is not yet clear how matching states should be identified. The state sample space may also be prohibitively large. We therefore assume that maintaining a database of all system states and attempting to recognize the current state is impractical. We therefore assume that π cannot be selected by λ as a function of the current state.

The system gain provided by each π policy is generally very state dependent. Figure 7.10 provides a typical sampling of 50 system gains for a G^2 policy and two QDP-based policies. These gains were derived for 50 randomly generated application states with 18 jobs, 6 processors and 5 operational modes per job. Such an application has more than $3.8e12$ possible mode vectors to choose from, yet we can see that all gains provided by these policies were at least 94 percent optimal with an average gain which was 98 percent optimal. But their relative performance differs from state to state. Even the relative behavior of the two QDP-based policies, QDP1 and QDP3, is not consistent, with QDP3 gain exceeding that of QDP1 in 8 percent of the states despite a less refined bandwidth quantum.

Let Λ represent the set of relevant policy-making policies, then the *optimal* policy, $\lambda^* \in \Lambda$, for a given application is that which selects the π meta-control policies which provide the highest *average system gain.* We assume that system gain is zero until the software meta-controller responds to each state transition, so that both gain and response time must be considered when seeking a meta-controller policy which maximizes the system gain. Section 7.4.1 begins by formulating the policy-making problem as a semi-Markov decision process, whereupon Section 7.4.2 provides a method for selecting an optimal stationary meta-controller policy when all system parameters, transition times, and state distributions are *a priori* known. Section 7.4.3 then introduces the original *follow-the-leader* policy devised for Markov renewal programs [FR73] and shows how it can be modified to provide us with an optimal *adaptive* policy for systems where runtime states and transition times are not a priori known. We end in Section 7.4.4 with experimental Monte Carlo results which show that our solution requires very few decision epochs to converge to optimal performance. This indicates that our modified *follow-the-leader* scheme can also be very effective when state distributions slowly vary with time. We end in Section 7.4.5 with concluding remarks.

7.4.1 A Semi-Markov Decision Process

We assume a finite set of system states S, representing given application and platform states which the software meta-controller must accommodate. We define a finite set of decision epochs, $T = \{1, 2, \cdots, N\}$, where each epoch represents a state transition which can occur at random points of time. We use x_t to denote the transition time at epoch t, i.e. the time lapse between epochs t and $t+1$. We also use the random variable $\mathbf{x}$ to reflect the distribution of transition times so that $\mathrm{E}\{\mathbf{x}\}$ represents the expected average transition time, assumed finite. When addressing long-running systems, we must therefore assume that $N \rightarrow \infty$.

Each decision epoch t requires an action a_t from a finite set of actions A, where each action represents a well-defined meta-controller algorithm $\pi \in \Pi$ for altering the software configuration to accommodate the new state, s. The selected action π takes $y_t = y(s, \pi)$ units of time to produce a software configuration with a given system gain, $v_t = v_\pi(s)$.

The subsequent reward r_t for that action is equal to

$$r_t = \begin{cases} v_t \cdot (x_t - y_t), & y_t < x_t; \\ 0, & \text{otherwise.} \end{cases}$$

The overall system gain for N epochs is then computed by

$$g_N = \frac{\sum_{t=1}^{N} r_t}{\sum_{t=1}^{N} x_t}. \tag{7.26}$$

A policy-making policy, λ, is defined by a sequence of functions $(\pi_1, \pi_2, \cdots, \pi_N)$, where $\pi_t \in \Pi$ determines the action, i.e. meta-control policy, to be used for decision epoch t. As previously mentioned, state identification is assumed to be impractical so we assume that π_t is a function of t only. A *stationary* policy is of the form $(\pi, \pi, \cdots)$, so that a stationary policy λ is one that always uses the same meta-controller algorithm for each epoch, while an *adaptive* policy can alter the algorithm used at each epoch t based upon past history.

7.4.2 Optimal Stationary Policy-Making

We now address the issue of an optimal *stationary* policy-maker when the runtime environment is *a priori* known. We assume that state transitions and actions are independent, so that the probability of entering state $s \in S$ can be determined regardless of the policy used. Letting $p(s)$ denote the probability that the system will be in state s, and letting $f_{\mathbf{X}}(x)$ denote the probability density function for transition times, then the expected reward for state s and the stationary meta-control policy π can be computed by

$$\mathrm{E}\{r_s^\pi\} = \int_{x=y_s^\pi}^{\infty} v_s^\pi (x - y_s^\pi) f_X(x), \tag{7.27}$$

where y_s^π and v_s^π are the response times and system gains produced by policy π for state s. Incorporating Equation (7.27) into Equation (7.26)

$$\mathrm{E}\{g_N^\pi\} = \frac{\sum_{s \in S} \mathrm{E}\{r_s^\pi\} p(s)}{N \cdot \mathrm{E}\{\mathbf{x}\}}. \tag{7.28}$$

Letting Π denote all possible meta-control policies, an *optimal* stationary policy, $\pi^* \in \Pi$ is that which maximizes the expected system gain $\mathrm{E}\{g_N^\pi\}$, i.e.

$$\pi^* = \arg\max_{\pi \in \Pi} \mathrm{E}\{g_N^\pi\}. \tag{7.29}$$

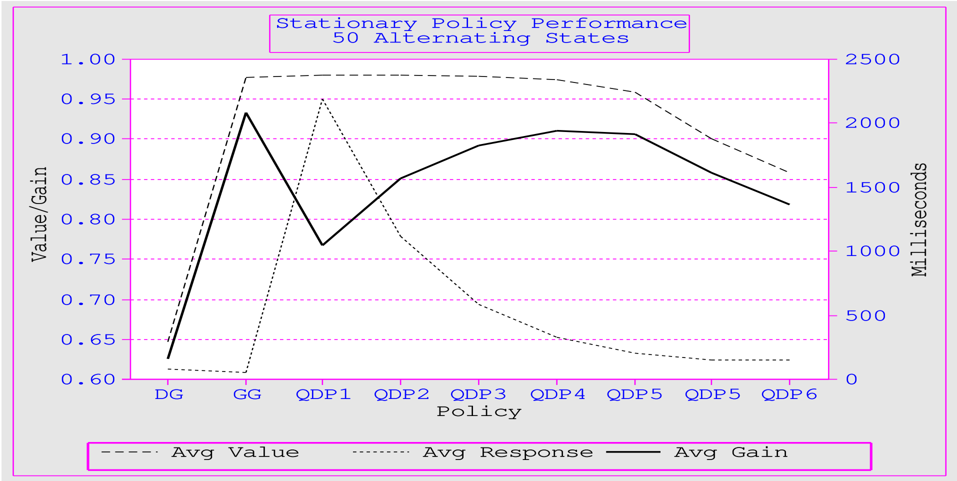

Figure 7.11: Stationary Policy Performance

Equations (7.27), (7.28), and (7.29) are readily computed, i.e. an optimal stationary policy is readily deduced, when all states, transition times and state distributions are known. But what can we do when they are not known?

7.4.3 Following the Leader

We now assume that the runtime states, S, state probabilities, $p(s)$, and transition time probability density, $f_X(x)$, are not known. Markov renewal programs have resolved a similar problem with an adaptive *follow-the-leader* policy [FR73]. This solution has several requirements which are not fulfilled by our problem, e.g. we cannot guarantee that any given state will repeat itself within a given number of transitions. Nevertheless, we contend that the basic elements of their solution can be successfully adopted because our problem is inherently simpler, e.g., because state transitions are action independent.

The primary *follow-the-leader* strategy is to sample the performance of all stationary policies in a manner which enables us to determine for each policy π the probability, $\hat{p}(\pi)$, that policy π is optimal. The *leading* policy, $\hat{\pi}_t$, at epoch t is that policy π which has the highest $\hat{p}(\pi)$. Long-running expected gain is maximized by letting the leading policy dominate the decisions. Using $g_N^{\hat{\pi}}$ to denote the system gain obtained by following the leader, the ultimate goal is to guarantee that:

$$P[\lim_{N\to\infty} g_N^{\hat{\pi}} = g_N^{\pi^*}] = 1. \tag{7.30}$$

In the Markov renewal problem, policy-dependent state transitions require that there be a certain state which must recur within a given number of transitions, whereupon a *cycle* is used to denote a sequence of state transitions starting and ending with that state. Specific cycles, called *forced* cycles, can then be used to sample the performance of non-leading policies. Forced cycles must be designed to ensure that there never be more than $\log N$ forced decisions over N epochs, so that their contribution to the overall reward and the overall time can be neglected. Thus the percentage of accumulated forced cycles converges to zero as $\hat{p}(\pi^*)$ converges to 1, and Equation (7.30) is maintained.

As previously mentioned, our particular semi-Markov decision problem differs in many respects with that of [FR73], and we can adopt only their philosophical approach. Most important, our problem is much simpler in that our state transitions are action-independent. This allows us to sample individual epochs rather than cycles, and it may also free us of the need for *forced* decisions. Here we must distinguish between meta-controllers with dedicated resources and those which use shared resources. When meta-controller resources are dedicated, then once a leader decision has been made, these resources can be used freely to explore the performance of other policies for the current state, until the next state transition[7]. When resources are shared, then we can allocate only a fraction, $\alpha(t)$, of resources for sampling the performance of the non-leading policies, subject to

$$\lim_{N\to\infty} \frac{\sum_{t=1}^{N} \alpha(t)}{N} = 0. \tag{7.31}$$

Our modified follow-the-leader scheme therefore comprises the following principles:

1. Each epoch, t, has a well-defined allocation of sampling resources for randomly exploring the behavior of non-leading policies for the current

[7] Experiments are aborted, of course, as soon as a state transition occurs.

state, s_t. In a dedicated meta-controller configuration, the computational time allocated is equal to $x_t - y_t^{\hat{\pi}}$. In a shared configuration, it is equal to a fraction, $\alpha(t)$, of the capacity β_c of a given cluster, c, and the available platform capacity for the application is updated accordingly. We note that sampled policies are selected randomly to avoid possible correlations between policies and the states for which they are sampled.

2. Each stationary policy $\pi \in \Pi$ has a two variables, $n[\pi]$ which counts the number of samples accumulated for π and $v[\pi]$ which estimates the total gain which would have been obtained using π for those samples, i.e. for each sampled epoch, t

$$v[\pi] = v[\pi] + v_t^{\pi}(1 - \frac{y_t^{\pi}}{\mathrm{E}\{\mathbf{x}\}}), \tag{7.32}$$

   where v_t^{π} is the sampled gain of policy π. We contend that $v[\pi]/n[\pi]$ serves as an very reasonable[8] estimator for g^{π}.

3. These estimators serve in choosing the leader $\hat{\pi}_t$ at each epoch t, i.e.

$$\hat{\pi}_t = \arg\max_{\pi \in \Pi} \frac{v[\pi]}{n[\pi]}.$$

   By virtue of the central limit theorem[BF81], these estimators gain a higher degree of confidence as more samples are accumulated. Thus $\lim_{N \to \infty} P(\hat{\pi}_N = \pi^*) = 1$.

Once again, we contend that the random sampling selection of π ensures that $v[\pi]$ reflects the probability $p(s)$ of being in each state, so that $v[\pi]/n[\pi]$ is indeed an reasonable estimator for g_N^{π}. We must also differentiate between cases in which $\mathrm{E}\{\mathbf{x}\}$ is known or unknown. In the latter case, we divide the current time by the number of epochs to estimate $\mathrm{E}\{\mathbf{x}\}$.

7.4.4 Experimental Results

Figure 7.11 provides the average system gains $\mathrm{E}\{v_s^{\pi}\}$, response times $\mathrm{E}\{y_s^{\pi}\}$ in milliseconds, and system gains $\mathrm{E}\{g_s^{\pi}\}$ for a given set Π of stationary

[8] A fully unbiased estimator requires complete knowledge of $f_{\mathbf{X}}(x)$ whereupon Equation (7.32) should be: $v[\pi] = v[\pi] + v_t^{\pi}(\int_{x=y_t^{\pi}}^{\infty} x f_{\mathbf{X}}(x) - y_t^{\pi}(1 - F_{\mathbf{X}}(y_t^{\pi}))/\mathrm{E}\{\mathbf{x}\}$. We note that the bias of the current Equation (7.32) converges to zero for $\mathrm{E}\{\mathbf{x}\} >> y_t^{\pi}$, which can be generally assumed to be true in the software meta-control context.

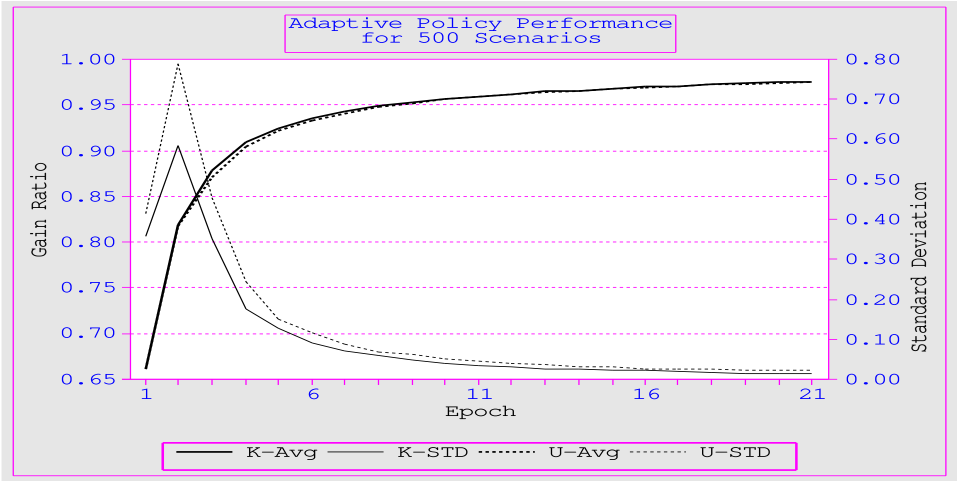

Figure 7.12: Short-Term Adaptive Policy Performance

policies and a given set S of states. As indicated, this set of policies includes the already mentioned G^{2}policy (labeled GG in the graph), several QDP-based policies (labeled QDP1, QDP2, $\cdots$, QDP6), as well as an inferior *density greedy* policy (labeled DG) which was included to introduce higher contrast. As implied by Equation (7.27), the average gain reflects a balance between the average system gain and average response as dictated by the transition time distribution, $f_{\mathbf{x}}(x)$. In our particular experiment, we used uniformly distributed transition times,

$$\mathbf{x} \sim U(0, 2.5 \cdot \max_{\pi \in \Pi, s \in S} y_s^{\pi}). \tag{7.33}$$

We also assumed dedicated meta-controller resources. The effectiveness of our adaptive policy was successfully verified for several random sets S of recurring application states with a uniform state probability, i.e. $p(s) = 1/|S|$. All results provided here present results for a single typical set S of 50 application states for a system with 18 jobs, 6 clusters and 5 modes per job. This set was tested for 500 different state transition scenarios and for each scenario we computed the adaptive gain $g_N^{\hat{\pi}}$ along with the optimal gain $g_N^{\pi^*}$ which would have been obtained by Equation (7.29), to obtain gain ratios, $g_N^{\hat{\pi}}/g_N^{\pi^*}$, for $N = 1 \cdots 1000$, as a measure of policy performance. Gain ratio

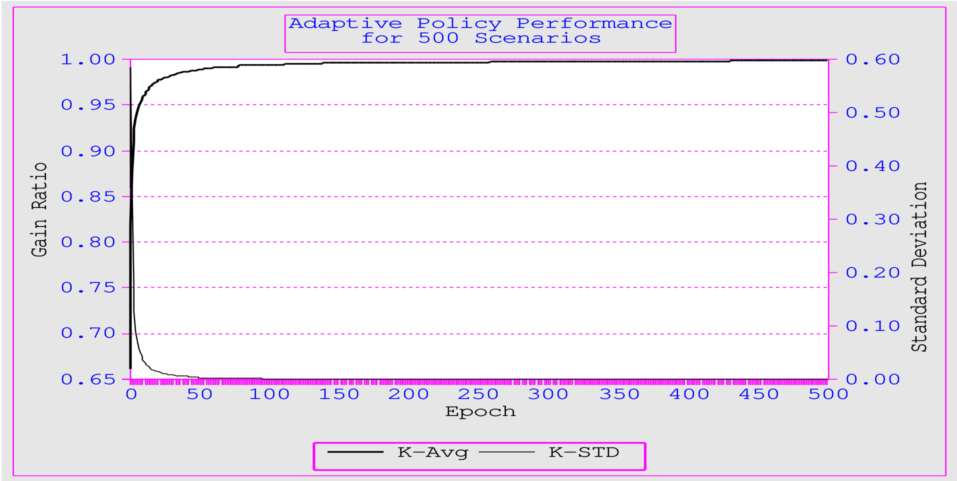

Figure 7.13: Long-Term Adaptive Policy Performance

averages and standard deviations were obtained to produce Figures 7.12 and 7.13, which show how our adaptive policy fared both in the short term and in the long term. The simulated average transition time, E$\{\mathbf{x}\}$, was a mere 10 seconds which is typical for many dynamic software applications. Even so, this provided us with a relatively high sampling rate, and allowed us to test several random policies at each epoch. The *K-Avg* and *K-STD* graphs in Figures 7.12 and 7.13 were produced when E$\{\mathbf{x}\}$ was assumed known. The *U-Avg* and *U-STD* graphs in Figure 7.12 were obtained by assuming that E$\{\mathbf{x}\}$ is unknown. We can see that our adaptive policy converges very quickly to the optimal for both cases, being already 90 percent optimal in less than 10 epochs. Within 200 epochs, our policy gain already reaches 99 percent and the gain ratio standard deviation converges to zero within 100 epochs.

Figure 7.14 shows us the leading policy distribution of $\hat{\pi}_t$ throughout the 500 scenarios. We can see that these distributions are fully correlated with the average gains depicted in Figure 7.11.

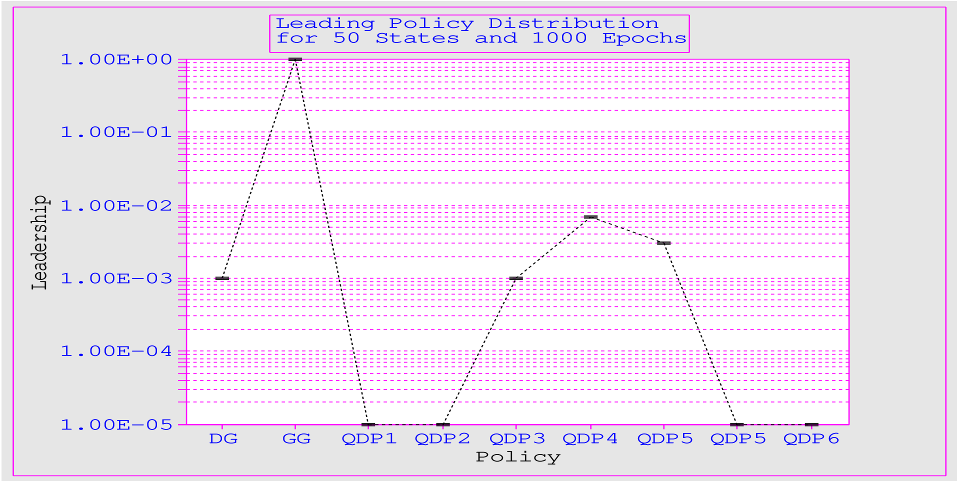

Figure 7.14: Follow the Leader Leadership Distribution

7.4.5 Concluding Remarks

As previously indicated, we can see that the adaptive follow-the-leader policy converges very quickly even for very short average transition times. This suggests that this policy can also be very effective when state distributions vary with time, as long as they remain stable for a significant duration. Given more extended transition times, one might also consider artificially introducing additional decision epochs for the most recent state to enable significant $\hat{\pi}$ leadership transitions to take immediate affect and thereby hasten gain ratio convergence. We also note that $\alpha(t)$ alters the platform state and therefore affects policy performance. These effects can probably be minimized by ensuring that $\alpha(t)$ convergence to zero is as gradual as possible.

7.5 Conclusions

Software meta-control decisions can be limited in scope, depth, and frequency to accommodate limited meta-controller resources and required response times. *Scope*, here, refers to the ability to maximize adaptations for only a partial set of jobs and processors. *Depth* refers to the ability to tradeoff higher system gain for shorter response times. A brute-force approach might carry out full-scale decision-making with each internal state transition, thereby considering all possible adaptations for *all* jobs, modes, and processors. If compromises must be made, this approach will usually sacrifice depth, e.g. a smaller q when using QDP. A more sophisticated approach, on the other hand, might resort to quick localized responses to state transitions, i.e. with little or no processor reassignment, combined with more relaxed periodic or externally-triggered full-scale adaptations. When dealing with a large number of required mode transitions, rather than have to carry them out simultaneously, we would like to explore the ability to devise sequences of mode transitions which can be carried out gradually, in a matter which can be coordinated with the internal job scheduling processes, without causing transient overloads in any of the processors. To further reduce the *frequency* of adaptations made, we might artificially bolster application bandwidths to accommodate anticipated state transitions which might occur in the near future. Work on these issues has only begun.

We recall that job mode bandwidths and rewards may be internally *state-dependent*, so jobs must notify the upper control layers when an internal state transition is about to occur. When optimizing system gain for a given situation, the system control architecture must also consider the dynamics of migrating jobs, changing modes, and other short-term consequences, such as a possible loss of system stability due to frequent adaptations. It is the *policy-maker*'s responsibility to consider all anticipated possibilities in determining *when* automated meta-control decisions should be made, which *method* should be used, what *scope*, i.e. which jobs, processors, and mode sets should be considered, and to what *depth*, i.e. how fast and accurate each decision must be. An automobile driver, for example, will always slow down as he approaches a crossing or any other situation which might require a sudden change of speed and direction. In a similar manner, the policy maker of the remote-control craft may also take precautionary measures, e.g. allocating a larger job bandwidth than necessary when a state-transition is imminent, or preparing a set of instant local adaptations for emergencies.

Thus defined, the policy making layer must clearly be tailored to the specific nature of each system. Nevertheless we anticipate that a well-defined set of evolving policy-making control models and solutions will eventually support enough applications to warrant packaging in a generic format. In a non-agile safety-critical system, for example, the policy maker might have to consider the probability of every possible state transition and the effect it would have on the job bandwidths to disqualify *sets* of job modes which could lead to a system overload before the system could adapt. The choice of which set of modes to disqualify should consider the anticipated impact on the system gain which would be obtained by the upcoming meta-controller decision. Any solution to this problem would be useful to all non-agile safety-critical systems.

Chapter 8

The ATLAS Case Study

ATLAS is *a* portable *top-layer audio synthesizer* which software synthesizes MIDI-directed music for arbitrary numbers of configurable instruments. Each instrument is provided with several alternate synthesis modes, each mode producing a different SNR (signal-to-noise ratio) *reward* and consuming a different pitch-dependent computational *bandwidth* to generate a steady stream of digital samples for that instrument. MIDI inputs dictate a very dynamic program state, which comprises the number, nature, and pitch of the concurrent notes which the ATLAS software must synthesize, mix, and output in *hard* real-time. The synthesis modes used to generate these notes are dynamically chosen to maximize the overall SNR of the ATLAS output for each program state, without overloading the available platform resources. Choosing *optimal* synthesis modes for each program state is NP-hard, so we employ the best-effort meta-control algorithms of Chapter 6. When run on any given platform, the average ratio between the obtained SNR and the optimal SNR for each program state is referred to as the system *gain*. ATLAS state transitions are quite frequent, requiring a new choice of synthesis modes several times a second.

Additional ATLAS requirements include support for a *soft* real-time interactive, menu-driven, graphic display, as well as a *hard* real-time recording of all events and decisions so that the timeliness of ATLAS activities can be validated and the system gain can be evaluated. Targeted platforms include several conventional parallel and distributed platforms which also have non-deterministic elements, so ATLAS has been designed to effectively accommodate these elements, as well. ATLAS is an excellent testbed for the study of complex real-time architectures because it is complex, platform requirements are minimal, it is readily configured and scaled to overload any conceivable platform, and hard real-time faults are readily detected by sensitive eyes and ears.

As described in Chapter 3, top-layer applications should run on any platform supported by an appropriate standardized virtual platform for real-time, VaPoR. Having targeted ATLAS for both PC-based and Indigo-base platforms, we have designed and fully implemented a VaPoR for both platform families, but only the PC version has been fully debugged. Thus our current ATLAS version is inherently binary-portable within platform families and portable at the source level between families, but it has thus far been tested only on PC-based platforms. It was initially developed on the author's 486 PC, where it ran correctly, but the note texture was generally poor. The application was then taken to the Pentium PC of the author's advisor and run without modification, and the notes automatically sounded much better. As we show in Section 8.3.1, a conventional design would require major software changes even when porting to a faster platform. In both cases, the choice of sample rates and synthesis modes was carried out automatically to maximize platform utilization without overloading it. Offline performance analysis confirmed that these automatic decisions were typically $\sim 95\%$ optimal. Furthermore, no audible faults were detected even though the allocated resources were assuming computation times which were significantly less than worst-case!

This chapter provides a detailed walk-through description of ATLAS development using a top-layer design approach. We begin in Section 8.1 with a quick overview of the ATLAS requirements to convince the reader of the realism of ATLAS as a test of complex, hard real-time computing. Detailed ATLAS specifications are provided by Appendix A. We then show in Section 8.2 how our top-layer approach was applied to ATLAS system design, also briefly discussing VaPoR.. For more details on VaPoR implementation please refer to Appendix B. Section 8.3 concludes with a comparative evaluation of the results, as well as suggestions for future ATLAS development.

8.1 ATLAS Requirements

ATLAS is a portable software application which synthesizes polyphonic MIDI-controlled music in real-time, for an arbitrary set of musical instruments. As depicted in Figure 8.3, ATLAS configuration inputs include wave-

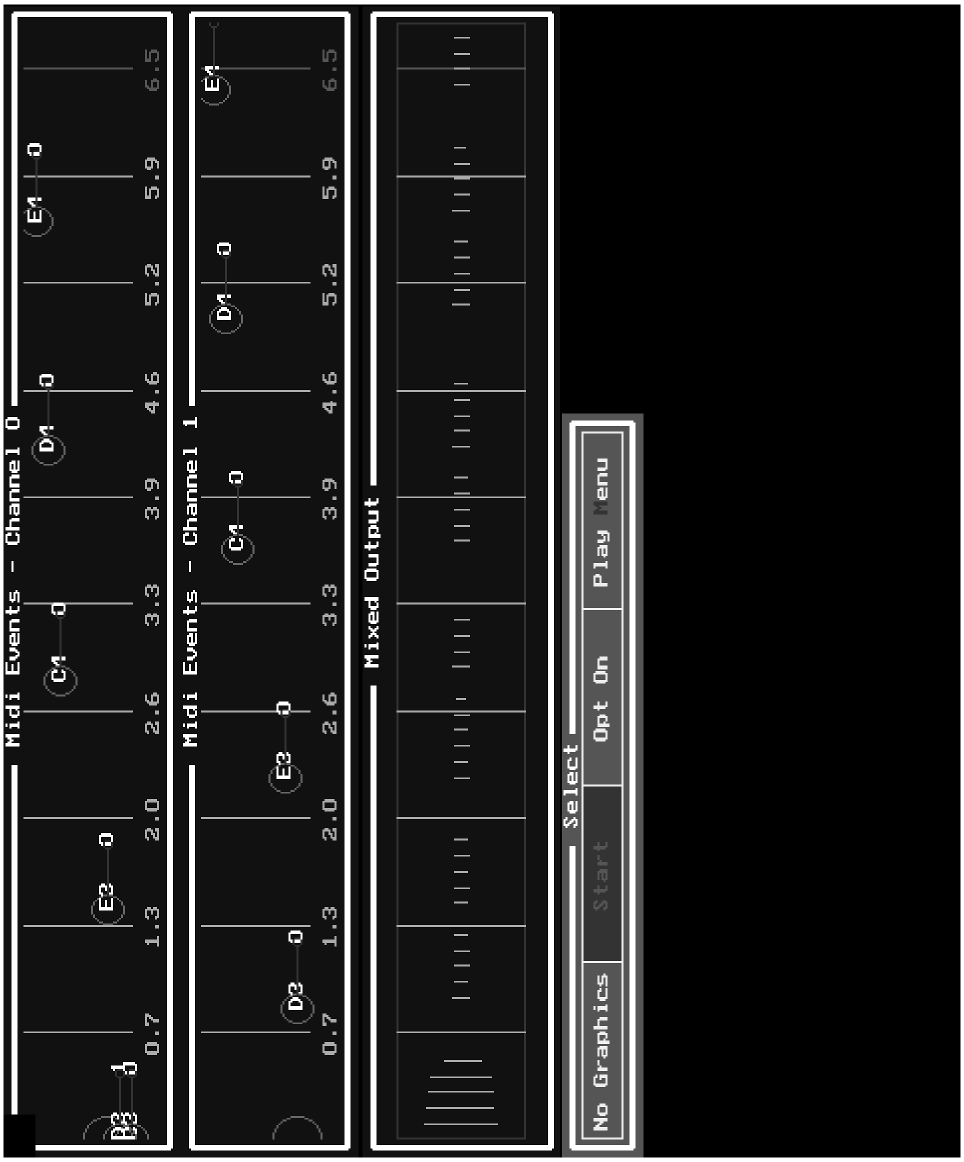

Figure 8.1: Real-Time Synthesis Output

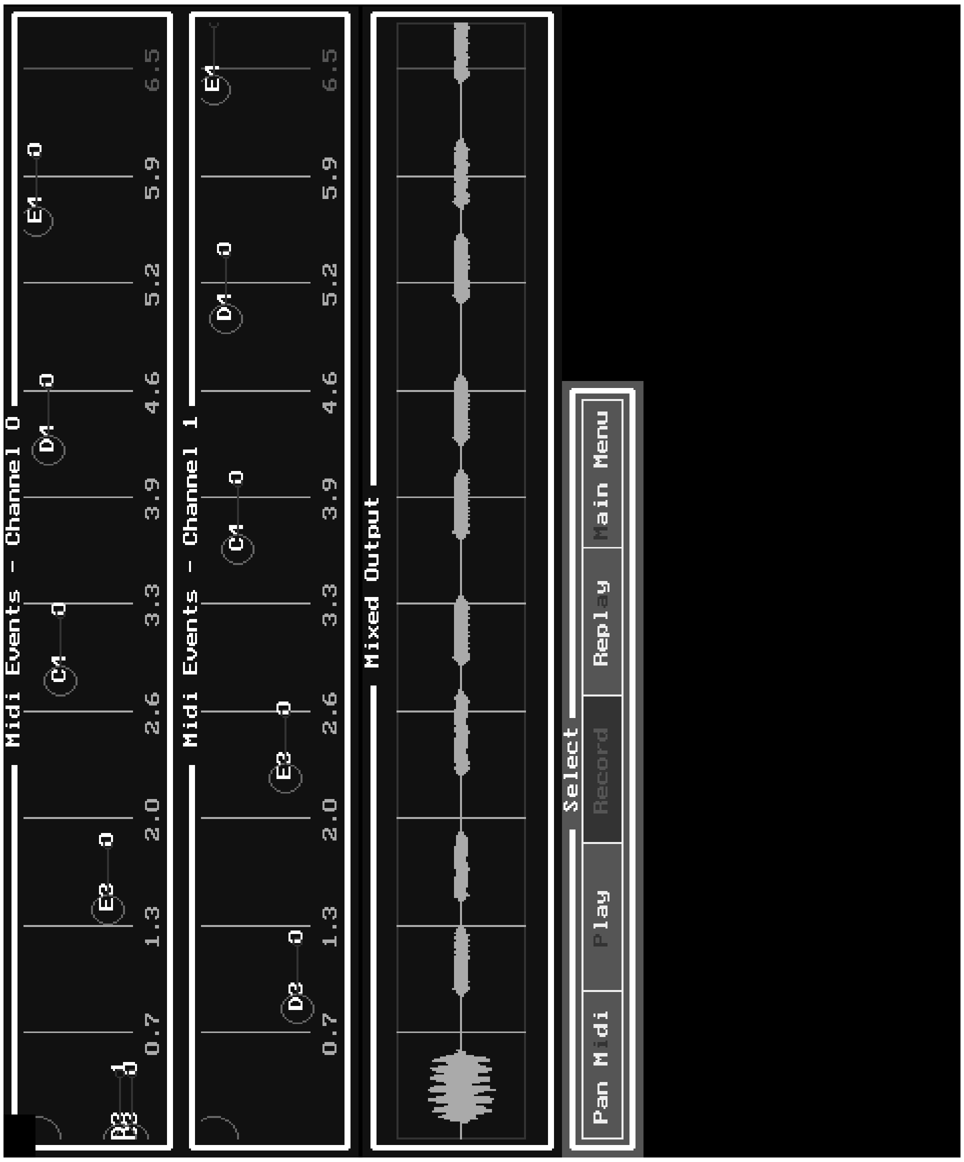

Figure 8.2: Platform Emulation Output

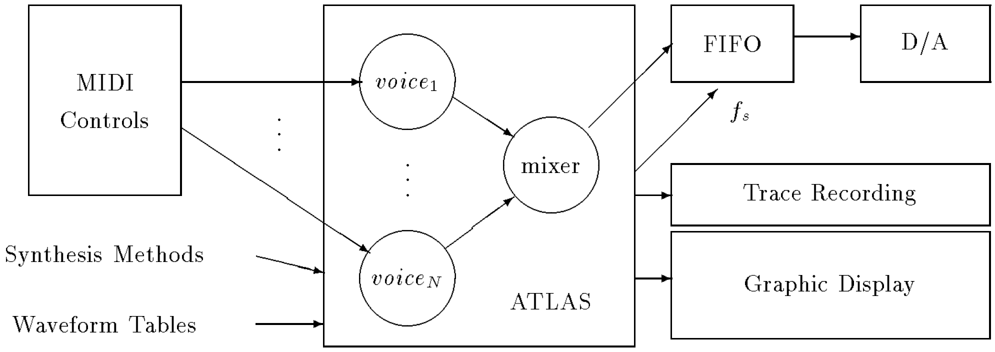

Figure 8.3: The ATLAS Application

form tables and synthesis methods libraries to customize the list of supported instruments and the sets of alternate synthesis modes for each instrument.

Dynamic online inputs include the stream of MIDI events which prescribe the notes which ATLAS must synthesize for each instrument, as well as an interactive GUI (graphic user interface) for activating available synthesis and display options. ATLAS outputs include a buffered audio output, i.e. the FIFO in Figure 8.3, a real-time graphic display of the notes being played and the audio output volume, (see Figures 8.1 and 8.2 in Appendix A,) and a recorded trace of all internal ATLAS activities so that ATLAS timeliness and gain performance can be offline verified and evaluated. All of these have real-time requirements or allow us to verify that the requirements are being met.

ATLAS *functional* requirements include

- handling MIDI inputs,
- carrying out the real-time musical synthesis,
- driving the audio output,
- maintaining an online display of MIDI notes and audio output,
- responding to mouse and keyboard input,
- and recording time-stamped events and system responses in a manner which can facilitate an offline evaluation of gain performance.

Note that most of these activities are periodic, but many of them are sporadic, and responses to operator input are aperiodic.

Discontinuities in the audio output, off-pitch notes, and off-beat notes, are all very annoying, even to the uneducated human ear, and are therefore not tolerable. Thus, *hard* ATLAS requirements include that all task deadlines which affect voice pitch, voice synchrony, and the steady flow of digital samples must always be met. The *soft* requirements are that the quality of the produced music be *best-effort*, i.e. that for every system state, the system must always select the synthesis modes that maximize the overall SNR (signal-to-noise ratio) without jeopardizing the completion of any hard deadlines. For any given platform, one can always devise a MIDI state, e.g. a large enough number of concurrent high-pitched voices with very demanding synthesis methods, which will overload that platform even when using synthesis methods with the lowest computational loads. Lower volume voices contribute less to the SNR, so lower volume voices will be shed until the hard deadlines of all remaining voices can be guaranteed. Additional *hard* requirements include the online recording of trace information, so that compliance with our requirements can be verified. Additional *soft* requirements include that the real-time display be maintained when the **graphics** option is on, and that the GUI always respond to operator events within a reasonable amount of time.

Why ATLAS?

ATLAS is an excellent case study for our research due to its complexity, availability, scalability, and verifiability. First and foremost, ATLAS must cater to task characteristics which have several dimensions of complexity: both hard and soft time constraints, periodic, sporadic and aperiodic tasks, fixed and dynamic activities. Secondly, it must cater to diverse platforms with varying numbers of processors and processing speeds. Thirdly, ATLAS is expected to produce best-effort sound quality for dynamic and arbitrary MIDI-dictated states, and it must do so without *a priori* knowledge of which instruments will be played, the number of concurrent voices, and at what pitch. Note that worst-case state requirements, e.g. 96 concurrent high-pitched notes, will generally be at least an order of magnitude greater than typical requirements. Last, but not least, ATLAS is required to run on conventional platforms containing non-deterministic elements with worst-case characteristics that may be an additional order of magnitude greater than typical.

Another very important ATLAS feature is platform *availability*. Hardware requirements for this application are minimal and readily provided for: an interactive graphic interface, any number of common D/A audio outlets, and a standard MIDI interface. As described in Appendix A, ATLAS can also be configured to run on platforms without audio outlets and MIDI interfaces, since these are readily emulated via standard MIDI and RIFF files. Graphical interface requirements are minimal and readily provided by any common PC or workstation.

ATLAS is also *scalable*, i.e. ATLAS configurations and MIDI-dictated inputs can be devised to overload any given platform, so that online load-shedding and mode selection policies can be evaluated for arbitrary overload levels. This scalability is achieved by providing synthesis methods of varying complexity, configuring different numbers of MIDI streams, playing musical scores with varying levels of voice concurrency, and selecting different levels of responsiveness via the configurable TimeFrame variable.

Last, but not least, ATLAS is verifiable, providing all the necessary information needed to verify its correctness and evaluate its gain performance. Besides the evidence provided by the trace recordings, hard real-time faults are readily detected by sensitive eyes and ears, making debugging and the assertion of hard guarantees much easier. We, therefore, highly recommend ATLAS as a testbed for researching all aspects of complex real-time software.

The reader interested in focusing only on the results of the experiment should skip to Section 8.3.

8.2 Top-Layer Design

In a top-layer design, all ATLAS activities must first be broken down into a set of schedule-independent subsystems called *jobs*, each job having any number of available *modes*, and each mode having well-defined resource requirements and gain contributions, which are respectively referred to as job mode *bandwidths* and *rewards*. Job sets, modes, bandwidths, and rewards may all be state-dependent. To fulfill the RtTS architecture requirements of Section 3.1, we must then choose appropriate time-sharing, meta-control, and policy-making policies. To fulfill the extended top-layer architecture requirements of Section 3.2, we must also devise bandwidth profiling policies, provide an appropriate VaPoR solution, and determine which program elements belong to the top layer, which to the portable PReTSL library, and which to the virtual platform VaPoR library. To these issues we dedicate

Task	min	max	avg	b_j
MIDI Input	0.000	0.029	0.003	
MIDI Display	0.000	0.053	0.004	
Mix & Audio Feed	0.130	0.139	0.131	
Mix & RIFF Record	1.851	2.733	2.371	
Trace Recording	0.001	0.023	0.002	

Table 8.1: Fixed Periodic ATLAS Jobs

the following subsections.

8.2.1 Decomposition into Jobs

All fixed (state-invariant) ATLAS activities are periodic. As listed in Table 8.1, these jobs include:

- MIDI input and display,
- mixing the voice outputs,
- either an output to the audio FIFO outlet with the real-time audio display or an RIFF recording with the platform-emulation display, subject to the current platform configuration, and
- the recording of all traced events and responses for offline verification and performance analysis.

Each of these fixed activities are treated here as a *job*, some of them having several *mode* alternatives for carrying then out. Also listed in Figure 8.2.1 are the typical and extreme execution times as measured on the author's 486/33 MHz PC over a limited sequence of time frames. We can see that these execution times may vary significantly from one time frame to another.

Besides these fixed activities, additional state-dependent activities include the voice synthesis activities directed by the MIDI input, as well as aperiodic responses to operator events. Each voice and each aperiodic response is also treated here as a job, so the number of ATLAS jobs is very dynamic. ATLAS requirements enable us to assume that the current set of jobs remains stable throughout each periodic time frame, with the time frame duration, T_p, being determined by the configuration file TimeFrame variable. If f_v is the current sample generation rate for a given voice, v, then the appropriate voice job must generate packets of $T_p \cdot f_v$ samples for each time frame. Packets generated by the voice jobs in each time frame are then mixed and output by the mix job in the subsequent time frame. The f_v sample generation rates and the synthesis methods used to compute these samples may vary significantly from one time frame to the next, so that voice job execution times are also extremely dynamic.

This job decomposition represents a good balance between parallelism and adaptability. On the one hand, it provides us with a large enough number of jobs to parallelize processing on a multi-processor platform, without overly complicating the management of those jobs. On the other hand, it also provides each voice job with several synthesis modes to choose from so that we can best-effort adapt to each program state and platform. Another conceivable breakdown would be to treat the voice synthesis of each MIDI channel as a single job, but this would unnecessarily limit parallelism. At the opposite extreme, one might suggest that each synthesis node be treated as a job to increase parallelism even more. This, however, would not fit our model because this would create precedence constraints between jobs, i.e., jobs would not be schedule-independent. This would also greatly increase inter-processor communication, greatly reduce the number of job modes to choose from, and create mode dependencies between jobs. Treating each voice as an independent job is therefore a natural and profitable choice for this application. Note, also, that all of these jobs are readily designed as single tasks, and can therefore be dispatched directly by the job time-sharing scheme, whereupon internal job scheduling policies are unnecessary.

8.2.2 Best-Effort Meta-Control

To maximize platform utilization for each platform, a top-layer application must be designed to promote enough flexibility so that an appropriate system gain function and meta-control policy will enable it to dynamically, best-effort, adapt to arbitrary program states and platform resources. The ATLAS application, for example, introduces this flexibility by providing for several synthesis modes per voice so that ATLAS can accommodate arbitrary MIDI-determined requirements and varying computational capacities. Each voice actually has two degrees of flexibility, a choice of synthesis methods, and an additional SRO (sample rate optimization) option for each method (details in Section A.3.1), if the opt on option is enabled. As described in Section A.3.1, when a voice note is turned on, all possible synthesis meth-

	decision (msecs)			performance per qtr			
Jobs	min	max	avg	1st	2nd	3rd	4th
4	0.0026	0.0042	0.0035	0.36	0.72	0.92	0.96
5	0.0035	0.0050	0.0044	0.27	0.77	0.96	0.98
6	0.0045	0.0063	0.0055	0.19	0.79	0.92	0.94
7	0.0059	0.0075	0.0066	0.21	0.86	0.94	0.95

Table 8.2: Typical Meta-Control Mode Profiles

ods, with and without SRO, are considered. When a note is already playing, only the current method, with and without SRO, can be considered. An additional *suppress* mode per voice job also enables us to silence the voice if necessary. Consequently, each voice job has several synthesis mode alternatives to choose from, each synthesis mode producing a different SNR reward and consuming a different execution time per time frame.

Multiple Meta-Control Modes

The selection of job synthesis modes for each time frame is carried out by an additional meta-control job charged with maximizing the total SNR without overloading the platform. To make these meta-control decisions, we have chosen the G^2 algorithm of Section 6.3 because of its lower overhead and because it is a sieve function which can always provide a valid solution whenever stopped. An arbitrary number of meta-control modes can therefore be attained by allocating a different maximum execution time for each mode. To experiment with this additional degree of flexibility, ATLAS supports several alternate meta-controller modes, each allowing a different maximum number of G^2 iterations to make these decisions.

As depicted in Table 8.2, the current ATLAS version supports four such meta-controller modes. Monte Carlo analysis is used to estimate the maximum number of iterations, N, required to make a non-constrained G^2 decision for varying numbers of voices, with Polyphony determining the maximum number of voices. Meta-controller mode, i, will then allow $N \cdot i/4$ iterations, with $i = 1 \cdots 4$. Extreme and average decision-making times are also profiled during the Monte Carlo analysis, whereupon these are multiplied by $i/4$ to obtain estimated execution times for each mode. Also computed are the average system gains obtained by each meta-controller mode in the

Monte Carlo analysis, so that these can be taken into consideration when choosing the meta-controller mode for each decision.

Studying the typical ATLAS values in Table 8.2, we see that *average* execution times are linearly proportional to the number of jobs, even though *maximum* execution times are somewhat greater than linear. Both of these findings are consistent with the worst-case timing complexity of Section 6.3 and with our findings in Section 6.4.3 that the actual number of iterations is usually much less that the worst-case. We also see that average non-constrained G^2 performance in our application does indeed reach the anticipated levels of 0.94–0.98. Moreover, these results indicate that halting G^2 after three quarters of the execution time will result in a performance loss of only a few percent, and that halting half-way through can still attain average performance levels of 0.72–0.86.

Safety Valve Modes

In addition to the above voice and meta-controller modes, several of the fixed jobs have also been provided with special degradation modes which we refer to as *safety valve* modes because they are only used in the event of an emergency, e.g. when the audio output buffer might deplete. Such emergency states may occur because our ATLAS version cannot determine job WCETs for each possible mode and state, and even if it could determine WCETs, we do not wish to assume WCETs which might be an order of magnitude greater than typical.

ATLAS currently has three safety-valves, one in each of the following jobs: the audio mixer, the audio display, and the trace recording. Audio mixing is degraded by effectively reducing f_s. Rather than mix all $T_p \cdot f_s$ samples in all of the voice output buffers, we simply mix every nth sample and duplicate it n times. This causes an effective reduction of the audio sample rate to f_s/n, also reducing the SNR. The audio display job simply skips a time frame, as does the trace recording job. Trace recording, however, is a *hard* real-time requirement, so the trace buffer is multi-buffered and the flushing of a buffer is skipped only if the chances of a buffer overrun are very remote. To verify this, and to analyze any bad calls if they occur, trace recording is always carried out in a manner which ensures that the number and timing of lost trace records will always be recorded.

System Gain Function

To maximize the overall SNR of the audio output, we have adopted the following system gain function,

$$v = r_{mc} \cdot r_{mix} \cdot \sum_{j \in J_{voice}} r_j, \tag{8.1}$$

where r_{mc} is the average meta-controller gain performance for the current meta-controller mode, r_{mix} is an SNR factor provided by the current audio mix mode, r_j is the SNR of the current synthesis mode for voice job j times the voice volume, and J_{voice} is the set of all voice jobs. Thus the system gain function considers the volume and synthesis mode of each voice, the current audio mixing mode, and the meta-controller mode which made the decision.

8.2.3 Breathing CE Time-Sharing

A primary constraint in our initial implementation is that we could not count on support for preemptive scheduling on our PC-based platforms. Fortunately, as already described in the beginning of Chapter 4, real-time job time-sharing is readily accomplished by a fixed, non-preemptive, CE (cyclic executive) dispatcher when all application tasks are *periodic* with periods which are harmonic multiples of a given minimal period, T_p. Such a fixed CE dispatcher can even be used in the presence of *sporadic* activities, as long as their bandwidths have been accounted for.

Thanks to the audio output queue[1], even soft *aperiodic* jobs can be dispatched, on-demand, by the CE, as long as they do not appear at frequencies which will cause the output queue to underflow. Letting C_i denote the computation time for periodic or aperiodic job i, N_i the minimal number of CE cycles between job i releases, and T_f the time it takes to empty the output queue, then no deadlines are missed so long as the average CE cycle time for all periodic and non-periodic jobs does not exceed T_p, i.e.

$$\sum_i \frac{C_i}{N_i} \leq T_p, \tag{8.2}$$

and as long as T_f is big enough to absorb occasional CE cycle times which exceed T_p.

[1] Note that Equation (8.2) is essentially the schedulability criteria for the preemptive RM policy when dealing with a harmonic task set, so the output buffer is necessary to compensate for the lack of preemptivity even when all jobs are strictly periodic.

job(s)	⋯	frame t	⋯
audio mix, feed & display		synthesis $t-1$	
voice synthesis		MIDI input $t-1$	
MIDI input & display			
meta-control decision		for frame $t+1$	
flush trace buffer to file			

Figure 8.4: The Three-Stage Pipeline

As previously described in Section 8.2.1, the number of ATLAS jobs to be executed in each CE cycle and the job computation times in each cycle are very dynamic, so we refer to our job scheduling scheme as a *breathing* cyclic executive, BCE for short. A similar approach has also been later adopted by PERC. Important BCE features include that it can accommodate periodic, sporadic, and aperiodic activity, that it efficiently exploits the inherent flexibility provided by the output queue, that it is completely deterministic when computation times are deterministic, that time-sharing overheads are negligible, and that it complies with the real-time time-sharing requirements of Chapter 4. The configuration file TimeFrame variable determines the T_p period to which all activities must conform, so all ATLAS jobs are assigned periods which are multiples of this period.

This non-preemptive BCE scheduling scheme applies only to the application tasks and it does not exclude the presence of underlying platform tasks which might be preemptive, e.g. an X server on the Silicon Graphic platforms. Note that we can always provide the platform tasks with a lower priority to ensure that they do not interfere with our real-time scheduling, and we can still ensure that these platform tasks, as a group, will not starve, by ensuring that the top-layer process does not consume the entire platform capacity.

Three-Stage Pipeline

All ATLAS jobs are BCE-dispatched in the order depicted in Figure 8.4. Here we see that each time frame t begins with an audio mix, feed, and display of the samples synthesized in the previous time frame. This is then followed by the synthesis of all voices dictated by the MIDI input of the previous time frame. Awaiting MIDI messages are then processed and displayed, a new meta-controller decision is made, and the trace buffer is flushed to the trace recording file. This essentially creates a three-stage pipeline, in which the MIDI input of each time frame is synthesized in the next time frame and actually output to the FIFO queue still one time frame later. This pipeline lets each meta-control decision determine which mode should be used in the following meta-control decision when it knows the number and nature of the voice jobs to be synthesized in the following time frame. It also simplifies future extension to parallel and distributed synthesis by maximizing the time available for transferring the voice output buffers to the mixer before these buffers need to serve again for then next synthesis cycle. Note that the worst-case MIDI response time is somewhat less than $T_f + 3 \cdot T_p$ because of the three stage pipeline.

Dynamic Voice Management

As described in Section 8.2.1, each voice job is invoked once per BCE cycle, so it must generate a *voice packet* with up to $T_p \cdot f_v$ voice samples per invocation. These voice packets are then mixed by the fixed Mix job and finally fed to the audio FIFO queue or to an RIFF recording. As described in Section A.2.2, voices are dynamic, but the maximum number of voices is fixed by the Polyphony configuration variable. These voice packet buffers may therefore be statically allocated to each voice before the real-time synthesis begins.

A more difficult buffer management problem is caused by the intermediate synthesis buffers required by the nodes in each program synthesis method. As described in Section A.1, a program library may contain any number of synthesis methods, each method comprising an arbitrary number of synthesis nodes, each synthesis node requiring an output buffer for samples which may serve as input to additional nodes within the synthesis method. Voice programming is dynamic, so any given synthesis method may serve up to Polyphony voices simultaneously, each voice having independent MIDI-dictated synthesis parameters, e.g. volume, attack, release, and pitch, so each synthesis method is pre-allocated a Polyphony number of synthesis parameter vectors. The memory consumption of these multiple vectors is relatively small, but a pre-allocation of multiple node output buffers could be intractable for many PC-based platforms. Fortunately, the non-preemptive nature of BCE enables us to get by with only one output buffer per synthesis node, since the contents of these intermediate buffers are of no consequence once the final voice packet is complete. Consequently, each pre-allocated voice packet can be dynamically linked to the synthesis method selected for that voice, each time the associated voice job is invoked.

Job Bandwidths

In a non-preemptive BCE dispatcher, normalized job bandwidths (see Section 3.1.1) are readily computed as $b_j = C_j/T_p$, when C_j has been measured on a processor, c, with the maximum normalized capacity, $\beta_c = 1$. Table 8.1 lists extreme and average bandwidths for each of the listed tasks, as randomly sampled on a 486DX/33 MHz platform with a TimeFrame of 0.1 seconds and a SampleRate of 8000 Hz. For a reasonably large sampling, we might assume that the maximum C_i times closely approximate WCETs. We can then readily determine that, when using an audio outlet, the worst-case bandwidth consumed by these fixed tasks is approximately equal to $\sum_j b_j = 0.243$. On a single processor, c, with a capacity, $\beta_c = 1$, this would leave us with a 76% capacity for remaining platform and dynamic ATLAS activities, e.g., mouse and audio interrupt handling, voice synthesis activities dictated by the MIDI input, and aperiodic responses to operator events.

The available capacity will, of course, be higher on a multiprocessor or more powerful uniprocessor platform. In a platform emulation mode, we could only emulate platforms with a capacity several times greater than that of the current platform due to the high Mix & RIFF Record bandwidth. Thus the *bandwidth* serves as a very useful measure for reasoning about computational requirements in a platform-independent way. Altering the configured TimeFrame may also have a significant effect on each of the job bandwidths. A larger TimeFrame will usually somewhat reduce the job bandwidths, but it will also require larger intermediate buffers and voice packets for voice synthesis.

Platform Noise

Many platform processes are interrupt-driven and their worst-case bandwidths must be accommodated. In our PC platform, for example, interrupt-driven processes serve the system clock, the mouse, the keyboard, the audio interface, and the cache-memory. The required bandwidths for each of these processes should be computed using the methods described in Section 4.2.2, whereupon these bandwidths can be deducted from the available platform capacity. Lacking the tools for doing this, we have chosen to treat these overheads as platform *noise*, using ad hoc profiling techniques to roughly estimate the average noise level to be deducted from the platform capacity, while making emergency provisions for sporadic bursts in the platform noise levels.

8.2.4 Policy Making

As defined in Chapter 7, the *policy-maker* decides when to activate the meta-controller, which jobs and processors should be considered, what algorithms to use, and whether higher optimality should be traded off for quicker response time. In ATLAS, many of these decisions have been resolved offline as part of the application design process, e.g. using G^2 in each time frame to decide for all voice jobs, and load balance them over all processors. The simple three-stage pipeline scheme enables the system to readily adapt to each meta-control decision at the onset of each new time frame. All fixed jobs require specific hardware devices, .e.g. MIDI input, audio outlet, disk files, so they are always assigned to fixed processors and their required bandwidths are deducted from the capacities of those processors.

Nevertheless, there are several policy-making issues which should be resolved online: how much time to spend in making each meta-controller decision, when to activate safety-valves so that no hard deadlines are missed, how to ensure that the trace recording safety valve is never activated when this might cause trace messages to be lost, and how to schedule aperiodic responses to operator events.

Choosing a Meta-Controller Mode

Referring back to Section 8.2.2, we recall that ATLAS currently supports four meta-controller modes to choose from, each mode representing a different tradeoff between computational requirements and the quality of the decision. This introduces a non-additive r_{mc} factor into the Equation (8.2) gain function which has to be reckoned with, with the four possible r_{mc} values being determined by Monte Carlo analysis as a function of current voice concurrency. The other non-additive r_{mix} factor depends upon whether the mixer safety valve will be activated during the coming time frame, this is determined by an independent mechanism, so the best we can do is compute an *expected* r_{mix} value as a function of the audio FIFO state and the spare platform capacities.

One way of dealing with these non-additive factors is to ignore them in computing the G^2 gradients, but use them in evaluating the result of each iteration. Each iteration produces a mode vector for all voice jobs, whereupon the platform utilization for that vector is evaluated to determine which of the four meta-controller modes would produce the highest gain for that mode vector in accordance with Equation (8.2). The new mode vector is subsequently accepted only if the new gain is indeed better than the gain of the previous mode vector.

Another alternative would be to use Monte Carlo analysis to independently determine the meta-controller mode as a function of the available platform capacity and possibly also as a function of the current voice concurrency. It stands to reason that making meta-controller decisions should never consume more than a certain percentage of available resources, and that quicker decisions may result in smaller gain losses when voice concurrency is higher. Our initial implementation, for example, automatically chooses the highest meta-controller mode which does not consume more than 5 percent of its capacity. For reasonably low noise levels, it is also reasonable to assume that the mixer safety valve will rarely be activated, so that all non-additive factors can be removed from the system gain function.

Safety-Valve Control

To maximize platform utilization, we wish to use typical execution times to compute the job bandwidths and then add relatively small bandwidth safety margins of a few percent to accommodate the common *noise* levels of the non-deterministic platform activities. We have yet to develop a model for choosing an effective safety margin, so our current ATLAS version uses the maximum sampled execution time within a very small sampling, which generally provided safety margins of approximately 20 percent, and is clearly significantly less than the actual WCETs[2]. All meta-controller decisions assume that the mixer, display, and trace recording jobs will run normally, and the bandwidths of these jobs are fully deducted from the processor capacities. The safety valves in each of these jobs are self-activated automatically if they detect that the audio output buffer might empty beyond an acceptable minimal level if subsequent jobs consume the maximum, rather than typical, profiled execution times. This minimal output buffer level adds an additional safety margin, realizing that the profiled maximum

[2]WCETs cannot be deduced by empirical testing, but even the maximum sampled execution times vary significantly from one run to another on the same platform, and we find in Section 8.2.5 that spikes in the underlying platform activity can be an order of magnitude greater than typical platform overheads. Thus, actual WCETs are significantly greater than the sampled WCETs on these platforms.

execution times are not the actual WCETs. Thus the chances of missing a hard deadline can be made negligible, even when generally assuming typical execution times, and even when the actual WCETs are not known.

Hard Trace Recording

Referring back to Figure 8.4, the first safety valve after each meta-controller decision is in the trace recording job which, when activated, will simply defer the flushing its internal buffer to the next time frame. The second safety valve, when activated, will degrade the effective mixer sample rate and the final safety valve will skip the audio display. Unlike these latter two safety valves, the trace recording requirements are *hard*, meaning that we cannot allow this safety valve to be activated when this may cause the trace buffer to overflow. The maximum number of trace messages per time frame is readily determined, so that the possibility of overflow can be evaluated before each meta-controller decision. When such a possibility exists, the trace safety valve is neutralized and the job bandwidth safety margins are augmented to ensure that the audio output buffer never underflows.

Aperiodic Responses to Operator Events

Our current ATLAS version does not require extensive aperiodic responses to operator events. Nevertheless, the ATLAS design should indeed have no difficulties in dealing with such responses. One way of dealing with aperiodic responses is to have the meta-controller determine in which time frame to carry out the response. The response could be treated as an additional job with a reward which rises with time to ensure that the response time will be limited. The number of concurrent voices may vary significantly from one time frame to another, so the meta-controller would naturally seek to activate it in a time frame in which the reduced SNR would be minimal. As previously described, the meta-controller might also consider the current FIFO state so that the aperiodic job will be activated when the FIFO can best absorb it. Ideally, responses to operator events should be broken down into a sequence of shorter activities so that the affect on the SNR will be less pronounced.

8.2.5 Job Profiling

The profiling of job execution times is currently accomplished by a Monte Carlo sampling, which is generally inappropriate for hard real-time systems

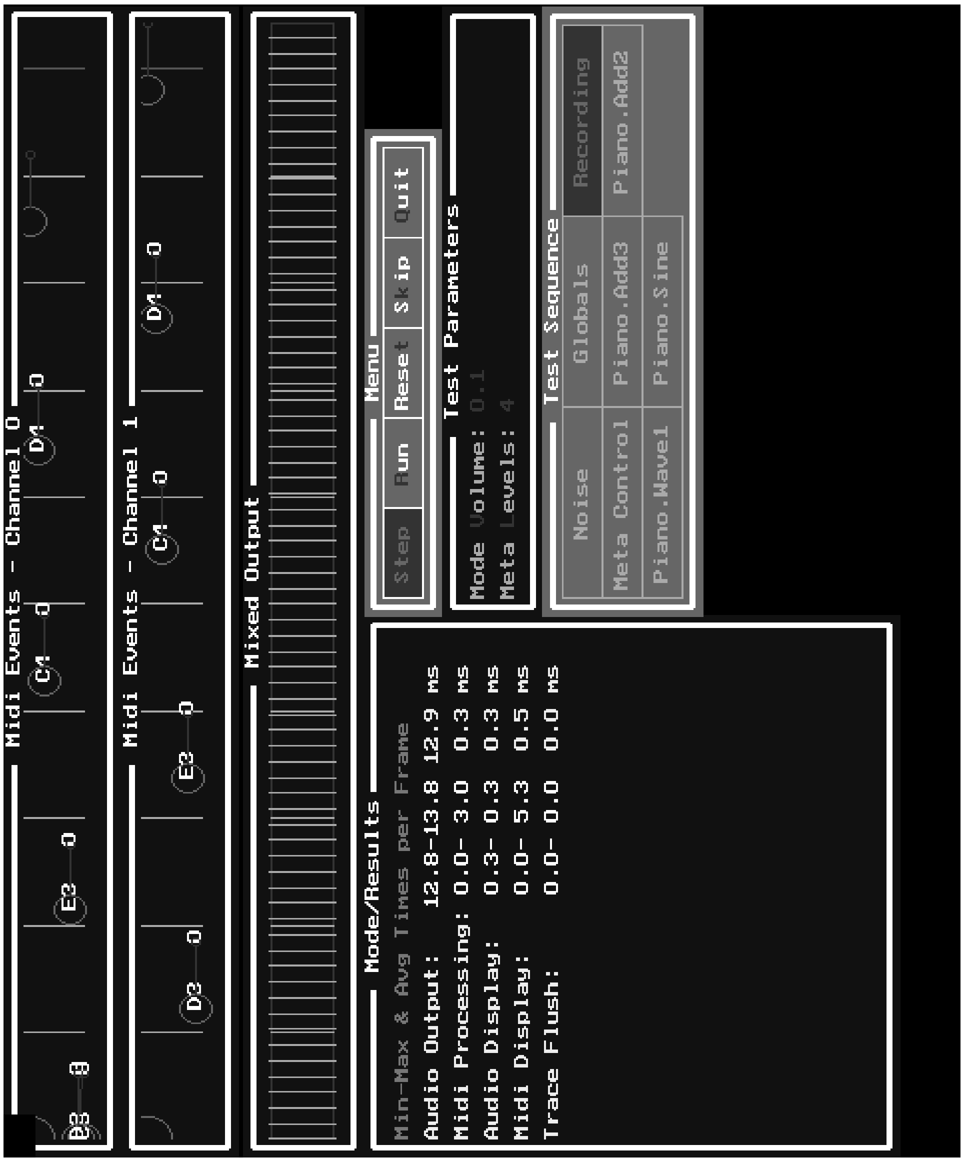

Figure 8.5: Fixed Job Profiling for Audio Output

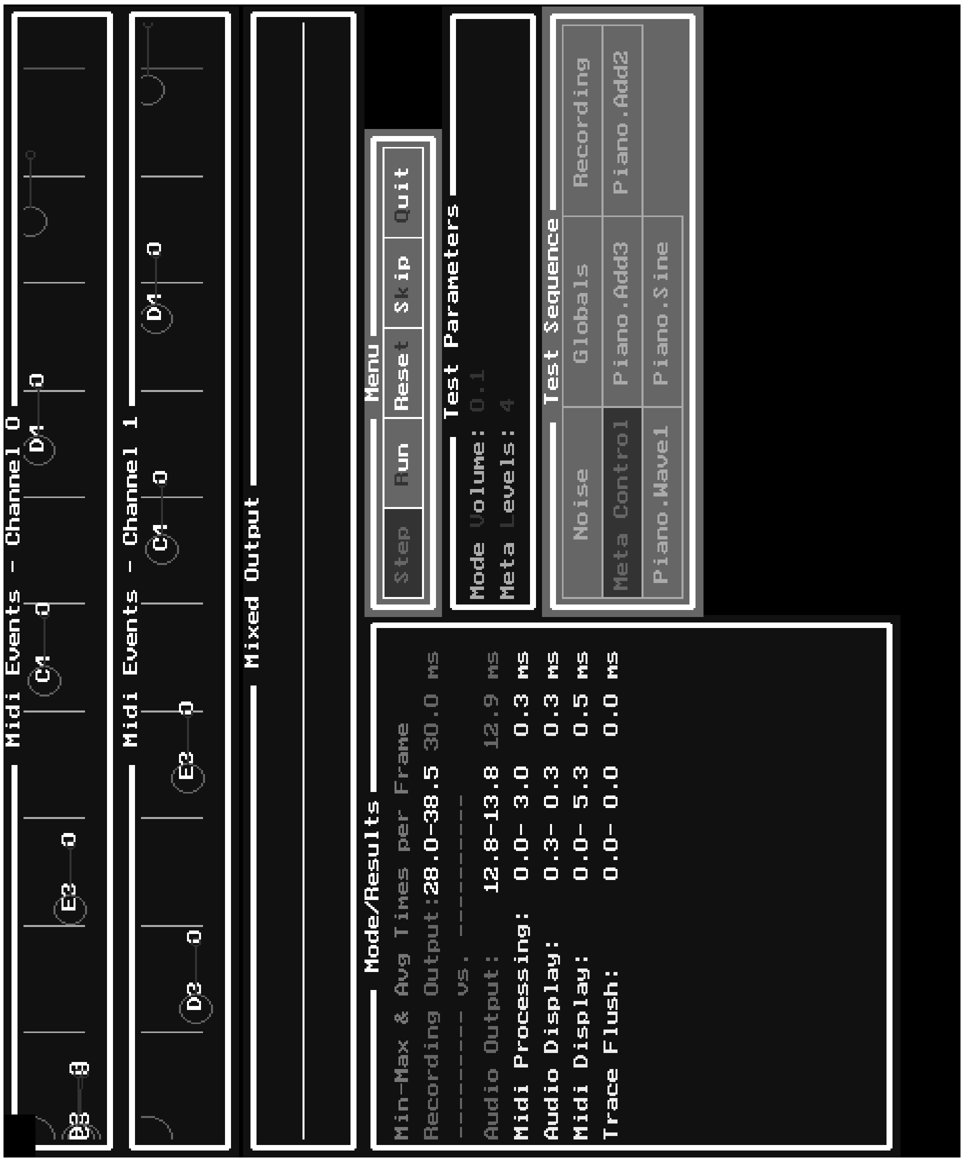

Figure 8.6: Fixed Job Profiling for RIFF Recording

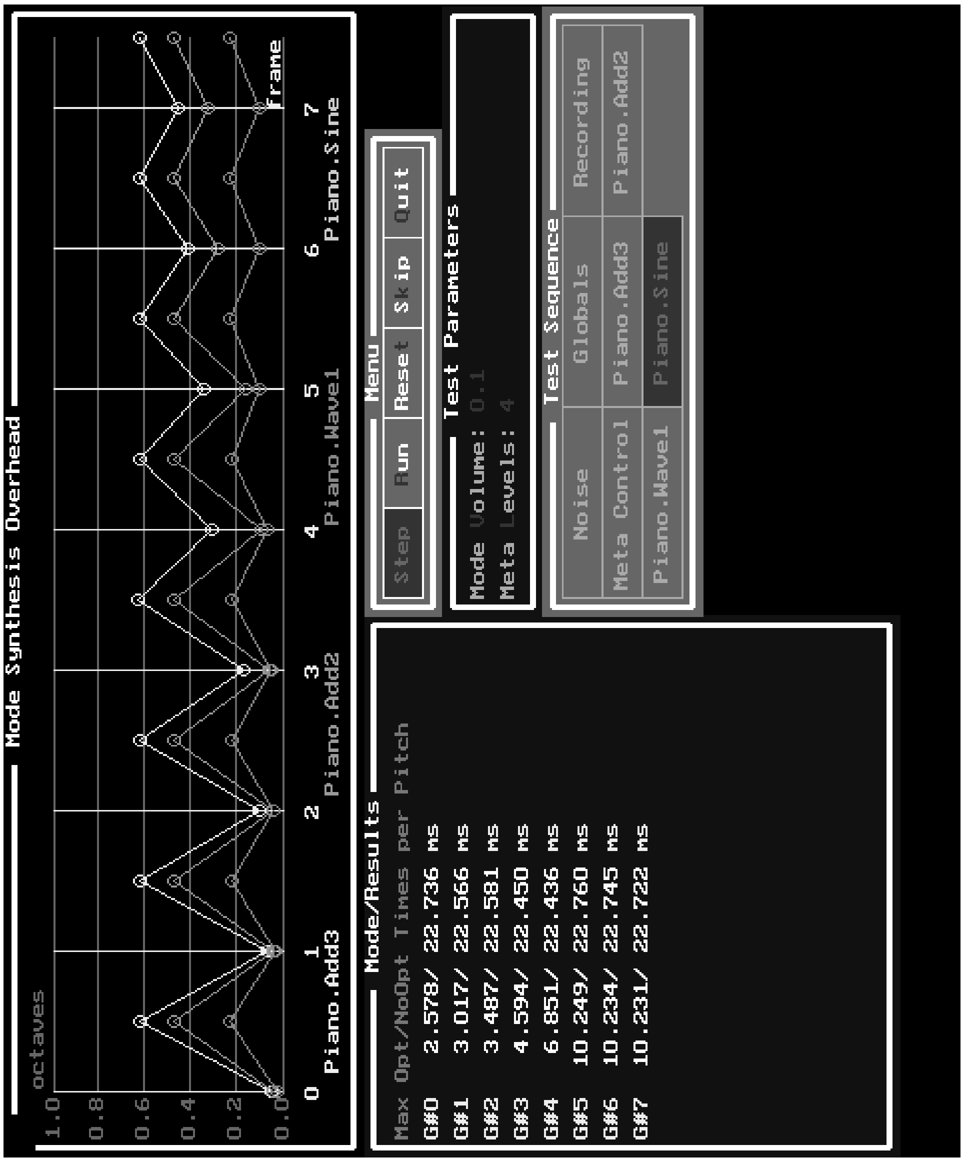

Figure 8.7: Synthesis Mode Profiling

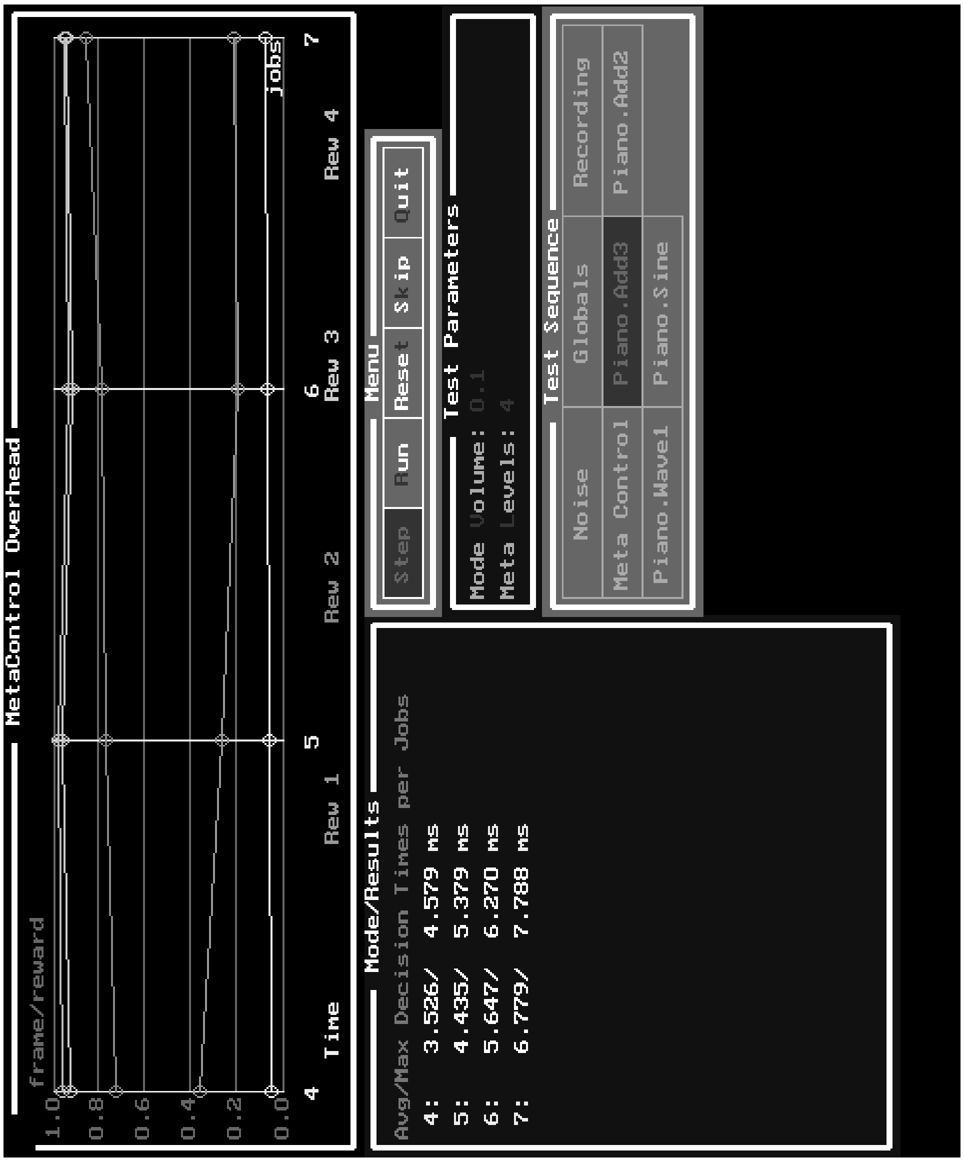

Figure 8.8: Meta-Control Profile

Figure 8.9: Platform Processes Profile

which prefer conservative WCETs over sampled sub-WCETs. Nevertheless, deterministic tools for evaluating job WCETs were not available, and the development of such tools was beyond the scope of our current research. To shorten the profiling process and to make life more interesting, the number of execution time samples was also kept relatively small, so that we can never safely assume that the maximum times were truly close to the WCETs.

Figures 8.5, 8.6, 8.7, 8.8, and 8.9 illustrate how job profiling is currently carried out. In Figure 8.5, we process a MIDI file without voice synthesis to obtain average and extreme execution times for all fixed jobs when producing an audio output. The same process is carried out in Figure 8.6 for all fixed jobs when producing an RIFF recording. As indicated by the latter, the difference between the two cases, 28.0–35.5 msecs vs. 12.8–13.8 msecs corresponds nicely with the disk access time. The estimates for MIDI input are, of course, only relevant for MIDI event rates which are similar in nature to those in the MIDI file used for profiling.

We then measure average and extreme execution times for voice synthesis as a function of pitch using all possible synthesis modes, with the results plotted and listed in Figure 8.7. Each plot represents a different synthesis method and the x coordinate denotes a specific pitch octave, contrary to the incorrect frame label which appears on the graph. Each synthesis method is tested with and without SRO, those with SRO being plotted on the vertical lines, and those without SRO appearing in between these lines. As one would expect, SRO mode execution times are significantly shorter than non-SRO modes, particularly in the lower octaves, and synthesis execution times with SRO increase with the octave number while those without SRO remain constant.

Meta-controller profiling is summarized in Figure 8.8. Each plot provided the average gain performance (reward) for a specific G^2-based meta-controller mode as a function of the number of concurrent jobs. The listed execution times relate only to the non-constrained G^2-based meta-controller decisions.

Platform Noise Profiling

Figure 8.9 illustrates profiler results regarding the underlying platform processes, also referred to as the platform *noise*. The current ATLAS noise profiler executes a periodic task which simply measures the time it takes to draw a line in the upper graphics window and records the time for statistical analysis. Each time measurement is referred to as a *cycle* time, and the lines drawn reflect the cycle times so that we can view their behavior. The minimal cycle times are assumed to reflect the actual task execution times, and anything beyond that is assumed to reflect additional *noise*, i.e. the platform overheads due to underlying interrupt-driven platform processes. The problem with this assumption is that the line drawing execution time is not constant, so that variations from one cycle to another can actually cause a snow-ball effect, since longer lines will amplify the actual noise. Nevertheless, it does provide us with some indication as to the nature and extent of the behavior. Also computed is the noise accumulated per T_p time frame. In the above profiler experiment we can see that we completed 29320 cycles over a period of 79 time frames. The minimal cycle time was 0.16 milliseconds, the average cycle time was 0.20 milliseconds, and the maximum cycle time was 85.7 milliseconds, which appears as a very distinct peak spike at approximately 5.2 seconds into the test. The average noise per time frame was 20.68 milliseconds, which is over 20%. In the vicinity of the peak spike the noise per frame even reached nearly 70%. Note that this is lower than the peak spike because the peak cycle did not fall within a single time frame and was therefore split between two adjacent time frames. All the spikes appeared when there was a burst of cached disk activity, so that more typical noise levels are those which appear in the beginning of the test. Such bursts are readily avoided on PC-based platforms by an appropriate DOS configuration. Tests carried out without cached disk activity but with plenty of mouse activity showed average noise levels of 3.9%, maximum noise levels of 6.6%, and a maximum cycle time of 2.24 milliseconds, Clearly, this maximum recorded noise level is not the worst case.

Referring back to the average and maximum job execution times in Table 8.1, we can see that the total average execution time per frame for the fixed jobs when using an audio outlet was only 14 milliseconds, while the maximum time totals were over 24 milliseconds. Some of these differences are attributed to varying MIDI input and display rates, and some is due to unstable levels of underlying platform noise. A major policy-making issue is to seek ways to distinguish between the two and to devise techniques for efficiently and safely accommodating them.

```
(00E968A4)  12824.733:FRAME Synthesize
(00E968F6)  12824.801:PLAY Frame Dur(95.99) ms
(00E9EA04)  12852.501:FRAME Mix Synth(27.70) ms
(00EA0249)  12857.710:FRAME Dealloc AudioMix(5.21) ms
(00EA029A)  12857.778:FRAME Sound Dealloc(0.07) ms
(00EA3162)  12867.819:SAMPLES Ready Fill( 1647) UserBusy(49.90) ms
(00EB0782)  12913.776:FIFO Irq Handled Pri(256.44) Dur(0.07) ms
(00EB0880)  12913.989:FIFO Ready Fill( 1279) FifoBusy(46.17) ms
(00EB09C3)  12914.260:FIFO Refilled   Samples(+  768) FlushFill(0.27) ms
(00EB0B5A)  12914.601:FRAME Flush AudioDraw(0.28) AudioProc(10.44) ms
(00EB0B83)  12914.636:FRAME Samples(768) Flush(0.03) Total(43.66) ms
(00EB0BA9)  12914.667:PLAY MidiLoop Dur(0.00) ms
(00EB0BE3)  12914.716:FRAME MetaCtrl MidiProc(0.05) ms
(00EB36E7)  12923.948:META ActualValue(4.876)
(00EB3755)  12924.041:META Level(3) Mix(On) Audio(On) Flush(On)
(00EB37A4)  12924.107:FRAME Synthesized MetaCtrl(9.39) ms
```

Figure 8.10: A Typical Recorded Trace Segment

Recording Profiler Results

Whenever activated from the main ATLAS menu, the ATLAS profiler will always record the results in a file named atlas.pro[3]. The maximum atlas.pro execution times will then serve as the basis for computing all job bandwidths in all subsequent ATLAS experiments. As illustrated by all of the above profiler displays, ATLAS profiling can be carried out automatically for all aspects of the system via the Run item, or we can use Test, Skip, and Reset to upgrade profiles for very specific aspects of the system. We can also edit the atlas.pro file manually to experiment with arbitrary profiles to be assumed.

8.2.6 Performance Analysis

To analyze performance, ATLAS supports a minimally intrusive trace mechanism whereby events, responses, and strategically placed program checkpoints are recorded in a cyclic trace buffer which is then flushed to an appropriate file for offline performance analysis. The timing of these trace file recordings is controlled by the meta-controller and the BCE dispatcher. We therefore use a double-buffered mechanism in which one cyclic buffer con-

[3]Actually, the profiled results are written to _atlas.pro to prevent automatic atlas.pro upgrades without an explicit user request.

tinues to record binary trace records while the other awaits being written out to the file, thereby enabling us to defer writing for a few time frames if necessary. In the event of a buffer overflow, the trace mechanism ensures that a special trace record is inserted which records the number of lost trace records.

Figure 8.10 depicts a typical trace segment after it has been translated into a textual format and analyzed by the ATLAS performance analyzer. As illustrated by the example, each trace id belongs to a specific class which is identified by the upper-cased name following each time-stamp. All FRAME and PLAY records serve to measure actual job execution times. FIFO records serve to monitor all FIFO queue related activities, including the audio device interrupts which control the emptying out of the FIFO queue via the audio outlet. META records provide us with the modes selected for each job by the meta-control so that we can analyze its gain performance. Whatever appears after the class name is derived from the current record value or from the difference between the current time stamp and the time-stamp of an earlier trace record. Thus the MetaCtrl execution time of 9.39 milliseconds in the trace record at time stamp 12924.107 milliseconds was derived by deducting the time stamp of the preceding FRAME record at time 12914.716 from the current time stamp. At time 12924.041 we find a META record whose value encoded that the meta-controller had decided that the next meta-controller decision would be carried out at level 3, which is the highest, and also indicating that none of the safety valves were activated, i.e. that the trace buffer was flushed, the mixer sample rate was normal, and the audio display was carried out. Together, these trace records provide all the information necessary to validate correctness and evaluate gain performance.

The offline evaluation tools have not yet been developed, so the ATLAS platform emulation mode has been temporarily modified to simulate the real-time synthesis mode while also using QDP with a very fine bandwidth quantum to produce ϵ-approximate *optimal* meta-controller decisions for comparison. Future offline evaluation tools will automatically make these ϵ-approximate decisions as part of the offline evaluation process, so that both real-time and emulation modes can be appropriately evaluated.

8.2.7 PReTSL and VaPoR Design

Part of the ATLAS challenge was to determine which application elements should be inherently portable, which should be tailored for each platform, how the various system components should be organized. As described in

Library	attributes/modules
PReTSL Tests (PT)	Monte-Carlo analysis programs
	PTANYPCK PTBINPCK PTPOLICY PTEPSILO PTPROFIL PTCONFIG
PReTSL Library (PL)	portable RTTS-related services
	PLJOBS PLRJOBS PLPLAY PLPROFILE PLMISC PLBINPCK PLGREEDY PLQDP PLMARKOV
VaPoR Tests (VT)	Monte-Carlo analysis programs
	VTCLOCK VTGRAPH VTMIDIB VTMOUSE VTPLAY VTPROGS VTRIFF VTWAVES VTALLOC VTSYMTBL VTDEMO
VaPoR Library (VL)	portable non-RtTS services
	VLGRAPH VLMENU VLTRACE VLMIDI VLMIDIB VLMIDIT VLRIFF VLNODES VLPLAY VLWAVES VLPROGS VLPARSER VLSYMTBL VLHEAP VLALLOC VLTYPES VLCONFIG VLUNITS VLTOKEN
VaPoR Machine (VM)	non-portable services
	VMCLOCK VMGRAPH VMSOUND VMSYSTEM VMAUDIO VMSBDRV

Table 8.3: ATLAS Libraries and Modules

Table 8.3, all PReTSL and VaPoR components are contained in three libraries. Portable RtTS-related elements which are not application-specific are incorporated into the PReTSL library. Portable non-RtTS services are incorporated into a VaPoR library. All non-portable services are incorporated into a VaPoR subset, called the VaPoR machine. Also provided are two set of test programs for the PReTSL and VaPoR modules.

All PReTSL library and VaPoR library modules comply with ANSI C++ standards, so that only the VaPoR machine modules need to be altered when porting to a new machine. The VaPoR test programs can then serve to ensure that they have been ported successfully. As illustrated by Table 8.3, the VaPoR machine comprises of only the following four modules: VMCLOCK for precision timing, VMSYSTEM for keyboard and mouse input, VMGRAPH for the graphic user (GUI) interface, and VMSOUND for the audio output. The VMAUDIO and VMSBDRV modules simply provide the respective lower-level Indigo and PC services for the VMSOUND module. Whenever possible, widely-supported standards were adopted, e.g. using the X11 *Xlib* library to facilitate VMSYSTEM and VMGRAPHIC services, and using the standard UNIX gettimeofday() to facilitate the VMCLOCK services.

As described in Appendix B, PC DOS lacks appropriate system support for almost all services provided by the VaPoR machine modules, so the PC VaPoR machine had to compensate by providing almost everything from scratch. Porting to an X11 environment was quite straight-forward because all we had to do was translate the internal lowest level services into those of *Xlib*. This ensures that the VaPoR GUI will have uniform behavior on all VaPoR platforms. It is, therefore, reasonable to assume that VaPoR can already be ported to a wide variety of platforms with very little effort.

As listed in Table 8.3, the portable services provided by the VaPoR library are very extensive. On the one hand, VaPoR covers almost all aspects of musical synthesis: RIFF and MIDI recording and replay, program and waveform library management, carrying out real-time synthesis, and providing translation functions for music-related units such as notes, frequencies, and sample table increments. On the other hand, VaPoR also provides several general purpose services, e.g. real-time trace support, efficient symbol tables and priority heaps, textual parsing tools, GUI menu management (VLMENU), and deterministic dynamic allocation schemes.

The PReTSL library contains all RtTS-related services for carrying out profiling, facilitating real-time time-sharing, and making meta-control decisions. All PReTSL classes were designed so that the same code would serve both ATLAS and the Monte Carlo evaluations of Chapters 5 through 7. According to the list of PReTSL modules in Table 8.3, PLJOBS provides JOBSET and PROCSET classes for managing the job parameter lists and current processor capacities required by the meta-control algorithms. PRJOBS also provides a MODEVEC mode vector class and a PROCVEC cluster vector class for containing the meta-controller outputs. To facilitate Monte Carlo experimentation, PLRJOBS inherits the PLJOBS classes and adds additional methods for randomizing parameters and capacities. PLBINPCK contains a variety of clustering algorithms for attempting to cluster a given MODEVEC decision over a given set of PROCSET capacities. PLGREEDY and PLQDP provide a variety of QDP, G^2, and DG-based meta-controller policies and PLMARKOV facilitates the Markov decision making process used for enhancing the QDP second-phase. PLPROFILE is responsible for job profiling, and PLPLAY facilitates real-time synthesis and platform emulation for arbitrary MIDI input files. All of the classes are designed so that arbitrary control policies can be incorporated and altered with minimal effort.

8.3 Conclusions

ATLAS was developed to evaluate the feasibility of developing top-layer, i.e. platform-independent, real-time applications which can best-effort adapt to dynamic run-time requirements and arbitrary platform resources. ATLAS real-time requirements are *complex*, including both hard and soft time constraints, periodic, sporadic and aperiodic tasks, static and dynamic activities. Targeted platforms include a variety of PC-based uniprocessor platforms, as well as parallel and distributed Silicon Graphics-based platforms. Thanks to it top-layer architecture, ATLAS should readily port to additional platform families. ATLAS was designed in a manner which should also eventually illustrate how a set of top-layer applications, e.g. several images of the ATLAS application, can run concurrently and reliably on a shared platform. Moreover, we endeavor to do this on conventional computing platforms which have not been specifically designed for *hard* real-time applications. ATLAS hardware requirements are minimal, including only a standard MIDI interface and a buffered D/A audio outlet which are readily emulated by a MIDI file and an RIFF recording. A versatile ATLAS configuration scheme enables it to be readily scaled to arbitrary work loads and to varying degrees of dynamic behavior and complexity. The atlas.pro profiler file also allows us to experiment with different safety margins regarding assumed execution times on each platform. ATLAS experiments are readily repeated to test a variety of approaches, and it has built-in facilities for validating its behavior and evaluating its performance. We know of no available applications which have all of these capabilities, nor do we know of available real-time platforms which provide the necessary facilities for developing such applications.

Our current VaPoR implementation has been fully implemented for two very diferent platform families, PC compatibles and Silicon Graphics, thereby demonstrating the feasibility of cross-platform virtual platforms for real-time. This has been more recently confirmed by the development of the PERC virtual machine for real-time Java programs. Thus far, ATLAS has been fully tested for a small set of musical scores and program configurations on two very different platforms within the PC family, the author's 486/33 MHz PC, and the advisor's Pentium which is more powerful by an order of magnitude. No audible faults were detected on either platform, even though assumed voice generation execution times were significantly less than worst-case! All trace records were recorded, and a detailed analysis verified that no *hard* requirements were breached. The best-effort average system gain, i.e. the SNR produced by the choice of voice synthesis modes for each time frame, was indeed 95 percent the optimal for each platform, clearly matching that which was theoretically anticipated by Chapter 6 and that which was empirically estimated by the meta-control profiler of Section 8.2.2. Above all, we have successfully integrated and applied the software architecture of Chapter 3, the real-time time-sharing principles of Chapter 4, the clustering algorithms of Chapter 5, the meta-control algorithms of Chapter 6 and the policy-making principles of Chapter 7. Though fully tested, so far, only on uniprocessor platforms, the extension to shared-memory multiprocessors is relatively straight-forward, and no difficulties are anticipated. Thus the current ATLAS version already provides ample evidence that a top-layer is feasible - it works!

8.3.1 A Conventional Approach

A conventional approach to such requirements would be to determine the worst-case MIDI state to be supported by the system as well as the worst-case set of operator-induced jobs, compute the WCETs for this state and these jobs, and seek synthesis methods for this state which could be accommodated by the platform. This is also how *hardware* synthesizers are designed, but, as previously described, this would result in a very low average utilization of platform resources, possibly as low as only a few percent. Furthermore, determining the WCETs for targeted platforms is not always feasible, particularly when dealing with the ATLAS-targeted platforms which have non-deterministic components, whereupon such an approach cannot serve *hard* real-time applications. A conventional approach would also require a complete worst-case characterization for all system components, including those of the underlying platform layers as well. On a UNIX platform, for example, we would also have to model all underlying audio, mouse, keyboard, display, cache management, and file I/O activities, as platform tasks which have to be scheduled by an appropriate integrated scheduling policy along with the top-layer tasks belonging to the application. When overloaded, conventional approaches shed loads on a FCFS basis, without being able to consider the effect of these loads on system gain.

Conventional approaches are also *inherently* non-portable. When porting from one platform to another, scheduling policies generally have to be revised because underlying platform tasks and all task characteristics will differ on the new platform, thereby altering the total set of tasks and the WCET ratios between them. This is true even when porting to a faster platform with identical machine instructions, since different operating systems, hardware devices, software drivers, and virtual memory configurations can all adversely affect scheduling. Moreover, different application task characteristics usually effect the way the program should adapt to varying circumstances, so conventional approaches cannot generally support automated best-effort adaptation in a portable manner.

8.3.2 ATLAS Advantages

ATLAS's RtTS architecture enables it to automatically adapt to arbitrary MIDI-specified musical requirements by selecting those modes which will maximize the synthesized SNR without sacrificing any functional and temporal requirements. ATLAS's top-layer design also enables it to automatically port from one platform to another, always best exploiting the resources available on that platform. Probably the most exciting aspect of our ATLAS experience is the success it appears to have had in more effectively accommodating the non-deterministic elements prevalent in conventional platforms. Our current version has been assuming typical, rather than elusive worst-case, execution times, with safety margins of only 20 percent. The output audio FIFO acts a shock absorber, and the current fullness of the FIFO queue acts as an indicator as to how close we are to a missed deadline. We can therefore relax our assumptions when the FIFO is full and be more careful when it gets empty. Meta-control decisions are made for each time frame, enabling us to quickly adapt when necessary. For more serious noise spikes, we also provide safety valves which can respond instantly, even before we reach our next meta-control decision epoch. The inherent flexibility and the adaptive RtTS architecture thus enable ATLAS to best-effort exploit the current platform while maintaining platform utilization levels which would probably be lower by two magnitudes if worst-case states and WCETs were assumed[4]. When unable to synthesize *all* of the MIDI dictated voices, ATLAS always sheds the less audible voices, rather than shed on an arbitrary FCFS basis. ATLAS is inherently portable. Therefore ATLAS performance will *automatically* improve with time as available platforms grow more powerful.

8.3.3 Future Work

Current ATLAS accomplishments are already substantial, but it is only a beginning. We have yet to carry out extensive performance analysis for a variety of configurations and platforms, including Indigo platforms from the Silicon Graphics family, as well. Here we anticipate that the platform noise for conventional Unix platforms will be greater, thereby requiring greater safety margins when assuming sub-WCETs, but the best-effort performance should be very much the same. We have yet to experiment with preemptive and multiprocessor platforms, to extend current meta-controller policies to deal with non-additive system gain functions, to incorporate analytic tools for worst-case profiling and for more detailed statistical profiles, and to develop more precise models for measuring platform noise and for exploiting statistical profiles to boost performance, Finally, we need more sophisticated policy-making schemes for dealing with short, medium, and long-term commitments which restrict our ability to adapt.

We would also like to significantly improve our VaPoR platform abstractions. Had we started the project today, for example, we might have chosen to use the Java or PERC virtual machines as a more practical basis for Va-

[4]We have yet to develop an analytic model for computing the MTBF, or the probability of missing a deadline, when using such a control mechanism on a given non-deterministic platform, but our current ad hoc model would appear to at least demonstrate how this higher platform utilization might be attained even for applications with hard real-time requirements.

PoR. VaPoR support for job preemption is also more readily achieved today as POSIX-like real-time multi-threading becomes more widely supported by conventional platforms. We therefore anticipate that standard technologies will eventually enable the implementation of more powerful and flexible VaPoR-like platforms with less effort. Future VaPoR extensions should also include support for distributed computing and virtual message-passing. As described in Section 8.4, the pipelined ATLAS design should somewhat ease the ATLAS transition to distributed platforms, as well, but we have yet to attempt this.

We would also like to show how conventional platform services such as dynamic memory management can be designed to be more deterministic, while services such as mouse input can be designed to provide various application-dictated levels of service which trade off higher response times for lower bandwidths. Such improvements can be very beneficial to real-time systems. It would be nice if real-time operating systems would provide us with mechanisms for trading off higher bandwidth for lower response times or higher memory requirements for each process. Thus a mouse driver should allow us to control the mouse baud rate which determines the maximum rate of mouse interrupts, and a DMA-driven audio interface should allow us to control the DMA buffer size to determine the maximum rate of audio interrupts.

Chapter 9

Managing Non-Determinism

The layered RtTS software control scheme of Chapters 3 through 7, is inherently deterministic. Even when using speculative scheduling policies within a soft or optional job, the real-time time-sharing scheme guarantees that deterministic scheduling policies within hard jobs will always receive the bandwidths necessary to meet all of their hard deadlines. Our current best-effort meta-control policies are heuristic, but they speculate only regarding the gain optimality of their chosen mode vector, and, as shown in Chapter 5, the clusterability of the chosen mode vector is always guaranteed if the sum of minimal bandwidths for the set of mandatory jobs does not exceed a known pre-determined level. Thus, the top-layer approach is clearly fully qualified for handling *hard* real-time systems when running on a *deterministic* real-time platform. The question raised in this chapter is whether hard real-time systems can run more *efficiently* on platforms which are *not* fully deterministic, i.e. one that exploits multi-level caching, execution pipelines, branch prediction, and instruction-level parallelism, to boost performance. It appears intuitively obvious that the only way to deal with such platforms is to always assume the worst-case, even if the WCET is one or two orders of magnitude greater than the typical execution time. This would result, however, in an extremely low, and probably unacceptable, level of platform utilization and system gain. What we suggest here is that there are hard real-time applications which can be top-layer designed to safely run on a non-deterministic platform, even when assuming sub-WCETs, provided that the behavior of the actual WCETs are known. Online software control then becomes a classical control problem: developing fail-safe feedback mechanisms which ensure that the system can *always* adapt to safe job modes when a system overload become imminent. Or even better still, use the feedback mechanism to ensure that a system overload will never occur. Our Chapter 8 experience with the ATLAS testbed has helped identify what those software control mechanisms might look like and what policies might be used to activate them. In this chapter we endeavor to suggest how these risk-management issues might be addressed more systematically.

We begin by briefly describing recent efforts to address non-deterministic platforms in a real-time context. As we show in Section 9.1, all of these efforts focus on more accurate techniques for minimizing the conservative assessment of WCETs. Section 9.2 then describes various ways to localize WCET assumptions so that they need be applied only to mandatory jobs in critical states. This leads to the definition of *secure* job modes in Section 9.3, whereupon we can define circumstances in which mandatory jobs may safely run in modes which are not secure. An example is provided by Section 9.3.1. Section 9.4 then lists the set of techniques which serve ATLAS to always meet its deadlines while assuming *sub*-WCETs. Conclusions and suggestions for future research are provided by Section 9.5.

9.1 WCET Prediction

Custom real-time platforms usually cannot compete with the cost-effective performance of conventional microprocessors. Unfortunately, it is currently impossible to accurately compute WCETs for real-time programs on these processors.

The prediction of a WCET is generally carried out in two stages. First determine the worst-case path through the code and then compute the WCET for that path. It is well accepted that the problem of determining the worst-case path is undecidable in general and equivalent to the halting problem. Several researchers have suggested restrictions on real-time programs to make this problem decidable, e.g. the absence of dynamic data structures, no recursion, and bounded loops. Even when the worst-case path has been determined, computing the WCET can be formidable for conventional microprocessors which use non-deterministic multi-level caching, execution pipelines, branch prediction, and instruction-level parallelism to boost performance. Pessimistic models can produce conservative WCETs which can be 30 times greater than actual WCETs [MW94], Thus the worst-case path cannot actually be determined until the WCET of each instruction has been determined, so this cyclic dependency leads to a pessimistic estimation of

WCETs. Additional difficulties stem from inter-task and intra-task interference, the former being caused by task preemption, and the latter by competing memory blocks belonging to the same task.

9.2 Relaxing WCET Assumptions

A variety of new analysis techniques for dealing with these problems have been recently suggested in the literature [MW94, LL94, BN94, LML+94]. The common goal of all of these techniques is to produce tighter upper-bounds for the WCET of each task. The hard real-time scheduler can then use these WCETs to guarantee that no critical deadlines will be missed. Unfortunately, even accurately estimated WCETs can be an order of magnitude greater than *typical* task execution times. This results in actual platform utilizations which are extremely low and wasteful, enabling us to harness only a fraction of the high cost-effective performance of the platform. What we seek, here, are methods for significantly raising the actual platform utilization without jeopardizing any of the critical deadlines.

Stankovic and Ramaritham [SR90] were probably the first to suggest that a WCET analysis might be necessary only for the worst-case scenario of critical tasks and for the *reactive* error handlers which are designed to bring the system to a safe state when an error has occurred. In earlier chapters, we have shown how the RtTS architecture enables us to *contain* WCET assumptions within specific jobs, modes, and states. Rather than *always* assuming worst-case scenarios in all jobs all the time, we can assume the worst-case only within critical jobs, and only in the states in which they are critical. What we wish to show now is that even within critical jobs in critical states we can often assume sub-WCETs, provided that appropriate mechanisms can guarantee that critical faults will always be prevented.

9.3 Sub-WCETs for Critical Jobs

Consider, for example, a critical job, j, with several M_j mode alternatives. Let *secure* mode, $\hat{m}$, denote a mode in which the job bandwidth has been computed assuming WCETs. Then any higher bandwidth mode, $m \in M_j$ may assume *sub*-WCETs in computing its bandwidth, subject to the following conditions:

1. An *emergency* adaptation from mode m to the secure mode $\hat{m}$ is known to be possible within a maximum mode transition time, t_m.

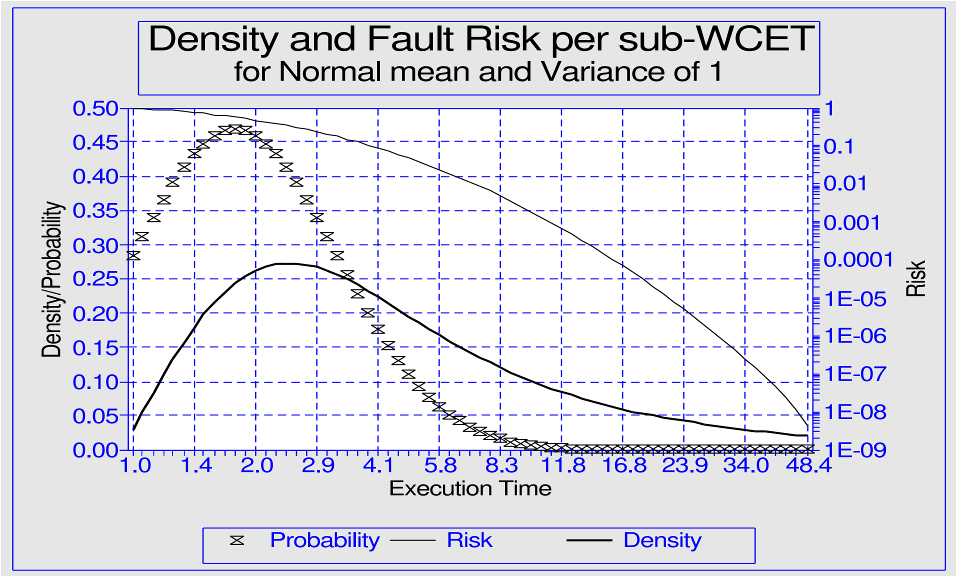

Figure 9.1: Fault Risk and Density per sub-WCET

2. An appropriate online monitor can anticipate all possible emergencies, i.e. timing faults, and can compute a minimal time to emergency, t_e.

3. An appropriate control mechanism will always trigger an emergency mode transition m to $\hat{m}$ when the minimal time to emergency, t_e, reaches t_m.

Each non-secure mode in each critical job must therefore have at least one secure mode for handling emergencies. Each non-secure mode may actually have *several* secure mode alternatives for handling different types of emergencies. The expected reward, $r_j(m)$, for each of these non-secure modes must also consider, of course, the expected mean time between emergencies, as well as the adverse effect of each emergency mode transition on the system gain.

Also note that statistical profiling might have been enough for ATLAS but a secure mode would have to use one of the analytic methods cited in Section 9.1 to compute deterministic WCET upper-bounds.

9.3.1 An Example

Consider, for example, a job consisting of n critical and identical periodic tasks with a common period, T, running on a processor with a capacity β. Due to non-deterministic platform elements, the task execution time is random, and therefore represented by the random variable, $\mathbf{t}$. The Probability graph in Figure 9.1 illustrates what a typical $\mathbf{t}$ distribution might look like, with best-case $t_{min} = 1$ and worst-case $t_{max} = 50$, and a near-normal distribution in between. The near-normal distribution was obtained by using a logarithmic x-coordinate scale, in which $x = \log_{\alpha} t$. Given t_{min}, t_{max}, and a list of sampled execution times, we can derive mean μ, standard deviation σ, and α values which closely approximate the $\mathbf{t}$ distribution. In Figure 9.1, for example, we assumed $\mu = \sigma = 1$ and $\alpha = 1.8$.

Once the $\mathbf{t}$ distribution has been established, we can compute the probability that $\mathbf{t}$ will exceed any given assumed sub-WCET, t. This is plotted by the Risk graph in Figure 9.1, which shows that the risk decreases exponentially with increased $(t - \mu)/\sigma$, as we are accustomed to in normal distributions. Let $p_f(t)$ represent the fault probability for an assumed t execution time, then $p_f(\mu) = 0.56$, $p_f(2\mu) = 0.12$, $p_f(4\mu) = 0.008$, and $p_f(8\mu) \simeq 10^{-4}$. The worst-case t_{max} is nearly 30 times greater than μ, but we can clearly assume $t = t_{max}/4$ to quadruple our performance, as long as we can reliably handle faults at an average MTBF (mean time between faults) of $T \cdot 10^{-4}$.

For a job with n tasks, the consumed job bandwidth would be linearly proportional to the sum of task execution time distributions, $S_n = n \cdot \mathbf{t}$. It follows from the Central Limit Theorem that for significant number n of tasks with independent execution times, the standard deviation of S_n will be reduced by a factor of $1/\sqrt(n)$. Thus for $n = 100$, we would have $\sigma = 0.1$, enabling us to achieve an MTBF of $T \cdot 10^{-4}$ by assuming only $t = 8\sigma + \mu = 2.6$, thereby attaining a twenty-fold increase in performance.

9.3.2 Emergency Thresholds

As described in Section 9.3, an online monitor would typically be responsible for triggering an emergency transition when a fault becomes imminent. To maintain the above MTBF of $T \cdot 10^{-4}$, we must ensure that the time to a possible fault never be less than an *emergency threshold* of $t' = 2.6 + \epsilon$, where ϵ is the time needed to respond in a manner which will avoid the fault. In ATLAS context, for example, the time to fault is equal to the number of samples in the audio FIFO divided by the sample rate, so that if 2.6 is the minimal time needed to refill the audio FIFO using a degraded mixer mode, and if ϵ is the maximum time needed to activate the degraded mixer mode, then we must ensure that the audio FIFO always has enough samples so that it would take $t' = 2.6 + \epsilon$ to deplete. Clearly the probability that this will happen is a function of the FIFO size, the initial FIFO state, and the nature of the **t** distribution. When assuming the actual WCET, i.e. $t = t_{max}$, then we need no emergency threshold, since the MTBF is infinite. Let MTBE denote the mean time between emergencies, then MTBE might be significantly worse than the MTBF, but it is the MTBE which determines how often we will respond. Note that in the ATLAS example, a large enough FIFO and an initial full FIFO state can cause the MTBE to be even better than the MTBF. In any event, when computing the job reward, we must take into account the MTBE-weighted adverse effect of these emergency transitions.

9.3.3 An Optimal Sub-WCET

As previously described, any assumed execution time $t \in [\mu, t_{max}]$ will produce a t-dependent MTBF, which can be determined as a function of the **t** distribution. To ensure that faults are avoided, we must determine an execution time threshold, $t' > t$ which will produce a t'-dependent MTBE. Assuming that T may vary and that the system gain is inversely proportional to T, then a greater t will raise the lowest attainable T on any given platform. Note that each assumed t will require a different job bandwidth, $b(t) = n\beta t/T$, on a processor with capacity β, whereupon the minimal T is equal to $n \cdot t$. Let $V(t)$ denote the system gain which would be obtained when assuming an execution time of t, then the actual gain for a $t' = t + \epsilon$ emergency threshold might be equal to $v(t) = V(t) \cdot [1 - p(t']$ where $p(t')$ represents the probability that the t' threshold will be reached. When assuming $t = t_{max}$, then $p(t) = 0$ and $v(T_{max}) = V(t_{max})$. Which t value would be considered optimal?

From our experience with greedy meta-control algorithms, it would appear that optimizing the job *density* might be worth while, whereupon the ratio between $v(t)$ and $b(t)$ should be maximized. Then the Density graph Figure 9.1 plots the job density which would be obtained in the above example for $n = 1$, assuming that an emergency mode transition can be carried out instantly, i.e. $t' = t$. The optimal $t = 2.4$ is only 2 percent that of the WCET, and it can only get better for larger n. This example demonstrates that the potential for boosting performance is indeed very significant. What remains to be shown is that faults can indeed be anticipated and that quick emergency transitions can indeed be attained. In the following section we use ATLAS to demonstrate how these issues might be resolved.

9.4 ATLAS Risk Management Mechanisms

PC WCETs can be much greater than average due to multi-level caching and due to a variety of interrupt-driven platform activities. Assuming WCETs would severely reduce performance. Not assuming WCETs would introduce known levels of non-deterministic behavior into the system, whereupon we would have to devise appropriate preventive control mechanisms to ensure that the system will always anticipate and avoid impending faults. Having to grapple with these non-deterministic elements within the ATLAS PC platform, we found that the top-layer architecture offers several new methods for safely accommodating these non-deterministic elements while predominantly assuming sub-WCETs.

ATLAS risk-management mechanisms contained the following key elements: *online monitoring* to anticipate impending faults, application-promoted *agility* and *safety valves* to minimize emergency transition times, multiple *shock absorption* mechanisms to maximize the MTBE, and mutual job *cooperation* to reduce the probability that an emergency adaptation will be necessary.

9.4.1 Fault Anticipation

The anticipation of impending faults is relatively simple in ATLAS, since the current FIFO fill indicates precisely how much time remains before a FIFO underflows. The FIFO fill is provided by sound.filled in Figure B.6, and the time to emergency is therefore equal to sound.filled divided by the sample rate, f_s. If the WCETs for all tasks are known, then we can readily determine at any given checkpoint how much remains to be executed and what its WCET might be before additional samples get fed into the FIFO queue. All mixer job modes are secure, i.e. they assume WCETs, so ATLAS always checks the FIFO fill before mixing the synthesized samples and feeding them, to assert that the mixer job mode chosen will always finish mixing the synthesized samples and start feeding before the FIFO underflows.

Other possible methods include monitoring slack time and using checkpoint monitoring to estimate actual noise bandwidths. Checkpoint monitoring can be automatically carried out by VaPoR when providing various platform services. Bandwidth preservation techniques [LSS87, SSL89] might serve to measure accumulated noise levels in this context.

9.4.2 No Risk Sub-WCETs

The ATLAS audio FIFO also enables us to assume sub-WCETs in a manner which will *never* require an emergency transition. ATLAS can always choose job modes with WCETs that do not exceed that of the time frame plus the time it would take the current FIFO to underflow, as long as the mean execution times do not exceed the time frame minus a certain dynamic margin. This dynamic margin is designed to ensure that the FIFO queue does not get depleted. Such a policy carries no risk whatsoever, since it essentially accounts for WCETs per time frame, but it achieves the same overall platform utilization that would have been achieved by assuming sub-WCETs. An adequately large FIFO would thus enable ATLAS to fully maximize the platform utilization. Note that a large audio FIFO essentially decreases the responsiveness of the system, so the FIFO size cannot be arbitrarily increased.

9.4.3 Promoted Agility

As illustrated by the above examples, the key to higher platform utilization is *agility*, i.e. the ability to quickly alter job modes to maximize the system gain in accordance with current available resources. This agility is facilitated by the top-layer architecture and its effective online meta-control algorithms, but it is up to the application designers to ensure that the online meta-control algorithms will have an adequate number of job modes to choose from. The hierarchical adaptive program model promotes additional mode extensions by ensuring that such extensions will not effect job modes already existing in the system.

9.4.4 Shock Absorption

"Firm" tasks are tasks with hard deadlines which are *occasionally skippable* [Lap93]. Thus the audio FIFO in ATLAS essentially converts the ATLAS synthesis tasks into *firm* tasks, which are still be treated by conventional methods as *hard* tasks, due to difficulties in exploiting this characteristic [KS95]. Our ATLAS example, however, demonstrates how this characteristic *can* be exploited in a top-layer architecture to extend the time available to make meta-controller decisions and to carry them out. As illustrated by

Section 9.4.2, the FIFO thus acts as a shock-absorber, capable of accommodating sudden bursts of noise without causing the system to crash.

Even without a FIFO, shock-absorption can be accomplished by an appropriate measure of surplus capacity which is reserved in each processor. This surplus bandwidth can be significantly less than that which would be needed to accommodate WCETs, but it must be large enough to provide enough time to switch to secure lower bandwidth modes when the surplus bandwidth gets depleted. Bandwidth usage must be monitored, rising overload probability must be anticipated, and appropriate emergency mode-transitions must be available.

9.4.5 Safety Valves

As already described in Section 9.1, WCET analysis can be difficult, and current techniques will often produce conservative upper-bounds which are an order of magnitude greater than actual WCETs. We define a *safety valve* to be any software component which can provide an immediate vent for sudden capacity overloads without requiring meta-controller intervention. The introduction of safety valves into the system can be very useful, particularly when WCETs are elusive.

In the ATLAS context, the mixer job is a classic safety valve, since it decides on its own how to degrade its performance in a manner which guarantees that the FIFO will not overflow. As described in Section 9.4.2, ATLAS would never need to activate a safety valve if all WCETs were indeed known. As described in Chapter 8, current ATLAS profiling is statistically accumulated, so that actual WCETs are not known. Another significant safety valve feature is that it can reduce the maximum emergency transition times, thereby lowering the sensitivity to noise by raising the threshold for triggering the emergency transitions.

Soft tasks in complex systems generally outnumber hard tasks by an order of magnitude. All soft tasks can act as safety valves by temporary self-suspension when surplus capacities get too depleted for comfort.

9.4.6 Job Cooperation

It has been long ago established by Kohelet that "two heads are better than one". WCETs are generally carried out per task, thereby introducing surplus bandwidth in each task. Overload probabilities for individual tasks and jobs are greatly reduced by time-sharing, since the risks of overloading are somewhat *diffused* amongst jobs sharing a given processor. The *central-limit* theorem, for example, indicates that if $n = 100$ tasks have equivalent variance in their execution times, then the variance of their total execution times will be reduced by a factor of $\sqrt{n} = 10$, so that surplus bandwidth can be reduced accordingly. As demonstrated by the example in Section 9.3.1, even a variance reduction of 50% can result in an MTBE reduction of an order of magnitude.

To facilitate such a technique, the performance profiler must differentiate between job-triggered transient overloads, e.g. disk I/O, and job-independent transient overloads, e.g. clock updates and mouse input. When dealing with job-independent transient overloads the same reserved capacity can serve any number of jobs sharing an given processor, c, so inefficiency due to worst-case assumptions is thus *diffused* among all jobs in J_c.

In another variation, each job distinguishes between hard and soft job bandwidths, and job load-balancing is carried out in a manner which guarantees that each processor has sufficient soft bandwidth to absorb the transient overloads. Work on these techniques has only begun.

9.5 Conclusions

The top-layer architecture presented in this thesis serves as an excellent framework for seeking an optimal balance between worst-case and typical execution times, between predictability and performance, between standard and specialized platforms. Some non-deterministic platform overhead might be modeled as independent *noise* with known worst-case bandwidths, whereupon they can be safely ignored by all application jobs so long as these noise bandwidths are deducted from the processor capacities. Adaptive applications can even *safely* assume sub-worst-case noise bandwidths if they can quickly adapt to lower bandwidth and secure job modes when monitored noise bandwidths approach dangerous levels. The adaptive and flexible nature of the top-layer architecture provides us with novel new ways to more efficiently accommodate non-deterministic platform elements without always assuming their worst-case characteristics. They also provide us with the means to exploit *firm* tasks, i.e. tasks with hard deadlines which are occasionally skippable. Work on these dynamic control mechanisms has only begun, but our experience with ATLAS has already indicated that they have practical potential.

Future research must develop methods for modeling the behavior of these control mechanisms. We also need viable methods for analyzing execution time distributions and for best exploiting their characteristics. We would also like to seek methods for profiling non-deterministic platforms in manners which enable us to determine which platform will best match the characteristics of a given application.

Real-time systems could then incorporate standard non-deterministic platforms to significantly boost system performance and reduce costs without threatening their safety. Rather than focus on elusive layer-by-layer determinism, top-layer designers will concentrate instead on determining the nature and extent of the control mechanisms required by a specific application to safely accommodate known levels of non-determinism within the platform and the environment.

Chapter 10

Conclusions

This concluding chapter provides a comparative evaluation of the strengths and weaknesses of our new approach, briefly summarizing our contribution. First, Section 10.1 contends that our top-layer approach is essentially a new paradigm for designing real-time software. We then discuss how well the goals of the top-layer approach were achieved in Section 10.2, current limitations in Section 10.3, and possible extensions in Section 10.4. Here, we show how the approach might be extended to distributed and fault-tolerant real-time systems, as well. Concluding thoughts and remarks are finally provided by Sections 10.5 and 10.6.

10.1 A New Paradigm

This thesis evaluates the effectiveness of a new paradigm for designing real-time software, capable of designing portable, dynamic, concurrent, and complex real-time applications which can automatically *best-effort* adapt to maximize a system-specific notion of performance called the system *gain*. The approach is referred to as a *top-layer* design approach, because it enables us to develop complex real-time applications with very little *a priori* knowledge regarding the platform layers.

We suggest that real-time software should be designed as collections of portable, state-dependent, *real-time* objects, called *jobs*. Each job encapsulates a proved, *offline*-determined, state-dependent, scheduling solution for a possibly complex set of internal job tasks when provided with a known *bandwidth* of platform resources[1]. Relatively simple *real-time time-sharing*

[1] These extensions sharply contrast with other object-oriented real-time extensions, e.g

mechanisms can then facilitate the sharing of resources between arbitrary sets of concurrent jobs, without jeopardizing any of the internal job schedules. For best-effort performance, jobs should also be designed for flexibility by supporting several *mode* alternatives, and by being able to readily shift from one mode to another, each mode representing a well-defined, state-dependent, tradeoff between required resource bandwidths and the reward contributed by that job to the system gain. Appropriate gain functions, online control models, and practical algorithms can then ensure that the system will *best-effort* adapt to maximize the system gain for arbitrary jobs states without overloading available platform resources. Arbitrary sets of concurrent, job-based, hard real-time applications can then reliably share arbitrary run-time platforms while best-effort adapting to varying run-time circumstances. The gain performance of such systems will therefore automatically improve as available platforms grow more powerful.

The notion of developing portable objects with self-contained hard real-time scheduling capabilities is new because *portability* and *hard real-time scheduling* are generally assumed to be a contradiction in terms. Hard real-time design, says a traditional paradigm [SR90], must be carried out from the bottom-up, *layer-by-layer*, requiring a full *a priori* characterization of all system components in all platform and application layers. The behavior of these components can then be modeled and analyzed *offline* to ascertain that a given task scheduling solution will guarantee all hard deadlines for all tasks in all possible system states. As described in Section 1.2, conventional design therefore commits to a specific platform relatively early in the development process. Thus, the real-time software is modeled as a set of tasks with platform-dependent execution times, and real-time scheduling is generally be resolved in a platform-dependent manner. Porting, even to a faster machine with the same instruction set, is rarely automatic because the assumed absolute or relative execution times are no longer the same.

Rather than focus on optimizing real-time scheduling to best accommodate a given set of tasks on a given platform, our new paradigm focuses on breaking down the real-time scheduling problem in a manner which will make real-time scheduling simpler, portable, more manageable, and less sensitive to change. Rather than focus on how to best exploit the characteristics of a specific platform, we focus on how to make the real-time applications more

that of MARUTI, which are designed to *reveal* and *export* detailed thread requirements and method execution times so that internal object activities can be appropriately scheduled by *external* control mechanisms.

adaptable and responsive to varying requirements and resources. Instead of seeking elusive layer-by-layer determinism, top-layer designers will attempt to determine the nature and extent of the control mechanisms required by a specific application to safely accommodate various levels of non-determinism within the platform and the environment.

10.2 Achieved Top-Layer Goals

The inspiration for our *top-layer* name and approach came from Stankovic and Ramamritham [SR90] who were the first to suggest that conventional layer-by-layer design constraints might have to be relaxed when dealing with next generation real-time systems. At the outset, all we wanted to do was to figure out how this might be realized and to show how major theoretical and practical issues might be resolved. However, as our top-layer methodology evolved, besides achieving our original objective, it became apparent that the job-oriented approach had significant advantages over conventional approaches to hard real-time design in the following major realms:

- in reducing online control *complexity*,
- in bolstering *scalability*,
- in boosting application *performance*,
- in enhancing *portability*,
- in supporting *concurrent* real-time applications on shared platforms,
- in reducing the high *costs* of real-time software development and maintenance, and
- in the *flexible* nature of the design *framework*.

These advantages are substantiated and discussed in the following subsections.

10.2.1 Reduced Complexity

A primary contribution of the job-oriented top-layer architecture is in its ability to address several formidable software control issues, simultaneously:

- the real-time scheduling of dynamic work loads with hard, complex, and varying task characteristics,

- the dynamic load-balancing of these work loads over arbitrary numbers of shared-memory processors, and
- best-effort adaptation to arbitrary sets of concurrent applications, running on arbitrary platforms within the targeted platform families.

As already discussed in Chapter 2, each of these issues alone is NP-hard. To simultaneously address dynamic *hard* work loads even on a single processor, PERC requires that all tasks be modeled as *simple* tasks, i.e. independent, periodic, and with fixed WCETs. To cater to hard workloads with more complex task characteristics, conventional approaches, e.g. MARS and Spring, will usually resort to *offline* planning techniques to devise *static* schedules which always assume worst-case task sets and states, with each platform requiring a unique solution. As illustrated by the AIMS example of Section 2.3, these pessimistic worst-case assumptions often result in failure to produce practical solutions for problems with which *ad hoc* methods succeed. For non-mandatory work loads, intolerable scheduling overloads can result from the highly complex acceptance tests needed to determine which incoming tasks can be assigned to a given processor without endangering the schedulability of tasks already resident. Task scheduling is complicated enough, so incoming work loads are generally accepted or rejected on a FCFS basis, i.e., automated *best-effort* adaptation is not even attempted. Thus the key to our success had to be a significant reduction in the online control complexities of each of these issues.

Several well-known techniques were used to successfully engineer this complexity reduction: encapsulation, abstraction, layering, and the reframing of the software control problem in a manner which reduces the *online* complexities and can be more efficiently resolved by practical, near-optimal algorithms.

Encapsulation – Rather than view concurrent real-time applications as a dynamic set of periodic and sporadic tasks with possibly complex task characteristics which have to be managed in the time domain, we *divide and conquer* to devise task aggregates, called *jobs*, with independent pre-packaged scheduling solutions so that arbitrary sets of jobs can share arbitrary processors without having to devise new scheduling solutions. As described in Section 4.2.2, each job essentially *encapsulates* the real-time scheduling complexities of each inter-dependent task aggregate within a periodic allocation of resources called a *bandwidth*, which is invariant for all platforms within a given family, each platform having an arbitrary number of shared-memory processors with arbitrary capacities per processor. Internal task scheduling complexities are thus hidden from other jobs and the higher level software control layers.

Abstraction – Another powerful technique is that of *abstraction*. The *mode abstraction* in our program model reduces complexity by letting the meta-controller consider many different forms of adaptation in a *uniform* manner. Altered scheduling policies, adjusted time constraints, different levels of imprecision, and polymorphic task sets, each job mode may represent an arbitrary combination of of these adaptation alternatives, each combination representing a different tradeoff of bandwidth and expected reward. VaPoR-like *platform abstractions* are essential to portability. Additional abstractions include the treating of underlying platform processes as a platform job and treating non-deterministic platform behavior as platform *noise*, thereby possibly freeing us of the need to seek new scheduling solutions for each application on each platform.

Layering – It is the job-oriented approach which also makes a *layered* software control model possible. As described in Section 3.1, the RtTS control model comprises several distinct control layers responsible for internal job scheduling, job time-sharing, job meta-control, and policy-making. The interfaces between layers are clearly defined in a manner which enables each layer to be developed and maintained independently. The *black-box job* paradigm of Section 3.1.4 greatly reduces the number and the complexity of the interconnections between the various software components and control layers. Thus the policy maker can choose and apply arbitrary meta-control policies without even considering the the scheduling policies being employed by each job. The choice of an EDF interleaver or a non-preemptive cyclic executive for job time-sharing is equally independent. This flexibility also promotes the organization of these policies in the PReTSL repository of Sections 3.2 and 8.2.7, so that each policy can be called upon when it best fits the needs of the application in appropriate circumstances. In a conventional approach, on the other hand, the algorithms used for task acceptance and scheduling are so inter-dependent that distinct, independent, layers are rarely possible.

Reduced Online Complexities – True, the overall complexity of real-time design has not necessarily been diminished, when considering the additional complexity of effective system decomposition into jobs, of seeking optimal internal job scheduling policies, and of computing bandwidths for all job modes and states. Nevertheless, we contend that a primary contribution of the architecture is that it encourages *offline* resolution of all of these additional complexities, so that the more critical *online* complexity of software control is substantially reduced. As shown in Chapters 4 and 5, online job time-sharing and job clustering is much simpler than online task scheduling on a multi-processor platform.

Practical Near-Optimality – It is the job-oriented view which facilitates a reframing of the software control problem in a manner which can be more efficiently resolved by practical, near-optimal algorithms. Appropriate job *time-sharing* schemes let arbitrary job sets reliably share any given processor in a given platform family if the sum of job bandwidths does not exceed the processor capacity. Thus, best-effort job load-balancing becomes a variation of the classic binpacking problem, with several practical approximate solutions. Chapter 5 evaluated the clusterability of these algorithms to determine their cost-effectiveness for typical situations and it also derived cost-effective variants for value-based load-shedding when overloaded. Dynamic job sets are therefore more easily managed in the bandwidth (frequency) domain, rather than attempting to dynamically manage complex task sets in the time domain.

Job time-sharing and the above *mode* abstraction translate the best-effort job mode selection problem into a a variant of classic zero-one knapsacking, which also has several practical approximate solutions. In Chapter 6, we have devised two families of practical meta-control algorithms for *simultaneous* best-effort job load-balancing and mode selection. A nice feature of both families is their ability to trade off response time for performance, and both have been sucessfully applied to the ATLAS case study.

Another aspect of our control model is that it also provides absolute measures for meta-controller performance. This enables Chapter 7 to determine that one of these two families is statistically ϵ-approximate and that an average performance of 95% is readily attained with reasonable overhead by both families of algorithms. The same chapter also shows how meta-controller performance can be estimated for a given application so that an optimal meta-control policy might be chosen. The ability to carry out frequent meta-controller decisions paves the way to *automated online* meta-control capable of catering to varying platform resources and *state-dependent* job modes, bandwidths, and rewards. And in sharp contrast with comparative efforts, job task characteristics can be arbitrarily complex, as long as scheduling policies and bandwidths can be determined offline for each job, mode, and state.

10.2.2 Scalability

Reduced complexity also contributes to the *scalability* of the approach. Job scheduling in the lowest control layer is independent for each job, so internal job scheduling overheads are clearly not effected by the number of jobs or the scheduling policies in other jobs. The EDF job time-sharing scheme of Section 4.1, of course, raises the internal scheduling overheads of the jobs, but the number of context-switches is unaffected. As shown by Section 4.1.4, the time-sharing overhead is of $O(\log N)$ complexity, N being the maximum number of concurrent time frames (activities) on the specific processor. Thus job time-sharing is inherently scalable, and, as illustrated by ATLAS, there are many cases in which job time-sharing overheads can be negligible.

As shown in Sections 5.1.1 and 5.1.2 effective job clustering for lower to medium capacity utilizations can be accomplished with minimal FF and BF complexities of $O(\mathsf{njobs} \cdot \mathsf{ncpus})$. For higher capacity utilizations, we would probably use FFD and BFD with additional complexity factors of $O(\log \mathsf{njobs})$. As shown in Section 5.2, combined clustering *and* value-based load-shedding can be carried out by our DG variants with a complexity of $O(\mathsf{njobs} \cdot [\mathsf{ncpus} + \log \mathsf{njobs}])$. As shown in Sections 6.2 and 6.3, even simultaneous resolution of load balancing, load shedding, and mode selection can be carried out by the QDP and G^2 algorithms with reasonable complexities for anticipated numbers of jobs and processors. QDP time complexity is equal to $O([\mathsf{ncpus} + \log \mathsf{njobs}] \cdot \mathsf{njobs} \cdot \mathsf{ncpus}/q)$ and lower complexities can be obtained by using less expensive clustering and mode vector sequencing policies. FFD-based G^2 time complexity is equal to $O(\mathsf{njobs}^2 \cdot \mathsf{ncpus})$ and lower complexities can be obtained by limiting the number of iterations with a sub-linear loss in performance. Thus, various levels of near-optimal meta-control are attainable with polynomial complexities which range from linear to quadratic.

The flexible program model and layered control model also contribute to scalability by supporting a *hierarchy* of jobs and meta-controllers. Job granularity is fluid, so a job may internally consist of several jobs managed by an internal meta-controller. Meta-controller decisions are also flexible, able to cover varying scopes of jobs and processors and able to do so with varying depth or precision. Thus our top-layer approach serves as a flexible framework for managing complexities at all levels.

10.2.3 Boosting Performance

Job time-sharing introduces additional scheduling *overheads*. Distinct bandwidths per job introduce processor utilization *fragments* within each job bandwidth which might have otherwise been exploited by other jobs. Moreover, as described in Section 3.1.1, when targeted for a diverse set of platforms, job bandwidths may also require additional buffering to accommodate non-linear differences in the platform execution time, thereby introducing additional bandwidth fragmentation. Nevertheless, we contend that these additional overheads and fragments can be minimized and that unexploited potential gains are insignificant when lacking practical means for exploiting them. Furthermore, we argue that these potential losses are essential to pave the way to substantially greater gains.

As described in Section 4.3, job time-sharing overheads are quantifiable, and they can be minimized by reducing the number of internal job scheduling deadlines and context-switches. As demonstrated by the ATLAS application, there are also cases in which internal job scheduling can be dispensed with all together. Unused bandwidth fragments can be minimized by seeking job task sets which can maximize the internal *bandwidth utilization*, also referred to as the *job utilization factor*, u_j, in the example of Section 4.2. As illustrated by that example, very high job utilization factors and very reasonable time-sharing overhead ratios are readily obtained with relatively simple task characteristics and task execution times which typically require scheduling decisions every few milliseconds. When dealing with more complex dynamic task characteristics and shorter task execution times, these job time-sharing overheads and bandwidth fragments will usually worsen, but integrated online real-time task scheduling alternatives would either be ineffective, intractable, or with complexities and overheads which would greatly overshadow those of job time-sharing[2].

But more important, the RtTS architecture paves the way to the follow-

[2]Thus, for example, complex acceptance tests, and online planning clearly incur very significant scheduling overheads. Additional bandwidth fragments are also introduced by pessimistic WCETs and the adjustment of task periods to obtain reasonable LCMs e.g. for cyclic executive and online planning schemes.

ing new ways to *boost* platform utilization and performance, boosts which may well over-compensate for additional scheduling overheads and bandwidth fragments.

Minimizing Worst-Case Assumptions – Best-effort meta-control enables flexible applications to automatically maximize the system gain online for each program state and for arbitrary platform resources rather than always assuming the worst-case regarding task sets, task state scenarios, and the available platform resources. Dynamic sets of essential hard tasks cannot be reliably accommodated by conventional methods, so worst-case requirements for the worst-case set of essential tasks must be *a priori* resolved and statically reserved, resulting in a tremendous loss of capacity utilization when the worst-case set of essential tasks and their WCETs are much greater than the typical. Such worst-case assumptions typically reduce the system gain by several orders of magnitude. Efficient top-layer support for dynamic, job load balancing enables assuming *actual* current job bandwidths rather than assuming worst-case job bandwidths for worst-case state scenarios. Thus, even when dealing with applications with relatively simple integrated task scheduling solutions, e.g. independent and periodic task sets, an RtTS architecture may be very worth while if these tasks have state-dependent, high-variance, execution times which result in WCETs which are much greater than the typical execution times.

Exploiting Non-deterministic Capacities – Nearly all computing platforms use non-deterministic techniques to achieve higher capacity, but conventional approaches cannot exploit these capacity gains in a *hard* real-time context within which strict WCETs are always assumed. Since it is often not practical to accurately predict WCETs on such platforms, real-time developers must generally reserve resources based on *conservative* upper bounds rather than *actual* worst-case requirements, making matters even worse. As described in Section 9.3, even in a hard real-time context, *agile* top-layer programs need assume WCETs only for essential hard jobs in their minimal bandwidth modes. For all other modes and all other jobs, we can generally assume sub-WCETs in a perfectly safe manner by monitoring current slack times and ensuring that appropriate emergency transitions to lower bandwidth modes are triggered when timing exceptions are imminent. As described in Section 9.4.2, adequate slack times enable us to obtain higher gain performance due to sub-WCET assumptions without even chancing emergency transitions. When *actual* WCETs are bound and known, these speculative techniques can actually provide *deterministic* guarantees for critical hard deadlines, making them appropriate even for safety critical systems. As described in Section 9.4.6, when actual WCETs are not known, WCET estimates can be less conservative since the risks of overloading are diffused by the job time-sharing. Thus, the top-layer approach may pave the way to exploiting yet untapped capacity potential within current conventional computing platforms.

Enhancing Schedulability – As described in Section 4.3, program decomposition into jobs may also take into account the different task characteristics to group tasks in a manner which enhances their schedulability. In RtTS, scheduling policies may vary from one job to another, from one mode to another, and from one job state to another, to best accommodate each job task set for each mode and state, e.g. to minimize internal task scheduling overheads and maximize the bandwidth utilization. Soft jobs can therefore employ speculative scheduling policies to maximize the expected job reward for those jobs without jeopardizing the hard guarantees provided by deterministic schedulers within other hard jobs.

Higher System Gain – In an overloaded situation, conventional load-shedding is carried out on an arbitrary FCFS basis, whereas our value-oriented load-shedding produces system behavior which is more valuable and less haphazard - and therefore more predictable. The RtTS architecture also enables us to allocate distinct bandwidths to soft and non-real-time activities to guarantee that these activities never starve, while choosing job modes which will guarantee that the hard deadlines are never jeopardized. Furthermore, these job bandwidths are allocated in a manner which reflects their relative importance, rather than always allocate higher priorities to the hard tasks. We know of no conventional approach which has these features.

Low-Cost Near-Optimality – All RtTS algorithms are inherently *near*-optimal so that the overall near-optimality depends solely on job design, e.g. the internal job utilization factors. Moreover, as illustrated by the ATLAS case study, their relatively low overheads make frequent adaptations and quick responses a very practical possibility.

Job time-sharing overheads and bandwidth fragmentation losses are therefore usually well compensated for by minimized worst-case assumptions, exploited non-deterministic capacities, enhancing schedulability, increased system gain, and the low-cost near-optimality of the control algorithms.

10.2.4 Enhancing Portability

As discussed in Section 8.3.1, conventional approaches are inherently *non-*portable. A top-layer approach maximizes platform independence. Only minimal assumptions are made regarding the target platform. The internal, offline-determined, real-time scheduling policy for each job assumes a dedicated *virtual* processor with a *virtual* capacity equal to the minimal job bandwidth so that reliable scheduling can be accomplished without a priori knowledge of the number of real processors and their capacities on the actual run-time platform. When provided with several mode alternatives, online *best-effort* meta-control algorithms ensure that system gain is maximized for arbitrary platform capacities. Reasonably low time and space complexities ensure that the RtTS software control model can scale well for larger number of jobs and processors.

As described in Section 3.1.1, the top-layer architecture is inherently *binary-*portable within a given family of platforms. It should also be *source-level* portable over all platform families which support the VaPoR virtual platform adopted by the top-layer application. As described in Section 3.2, generic PReTSL monitoring and profiling free the jobs of the need to provide platform-dependent task characteristics. Our ATLAS experience, previous Concord experience, and more recent PERC experience has shown that such virtual platforming, monitoring, and profiling is quite feasible. More advanced profiling will eventually help determine which platform will best match an application's needs when evolving requirements and constraints have been finalized. Thus, appropriate VaPoR implementations and portable PReTSL libraries should facilitate low-cost porting of top-layer applications when platforms are expanded or replaced to deal with inadequate performance, growing requirements, and obsolete hardware. System gain will therefore automatically improve as technology advances and platform capacities increase. Once portability is achieved, any given real-time problem should ultimately disappear as computers grow faster and more powerful.

10.2.5 Supporting Concurrency

Concurrent real-time applications are traditionally treated as a dynamic set of real-time tasks which must be appropriately scheduled by a uniform scheduling policy. As outlined in Sections 2.2.1 and 2.2.2, several practical approaches to this problem have been developed and implemented, e.g. Spring, MARUTI, ARTS, and PERC, but such solutions are either limited regarding the complexity of task characteristics that can be catered to or they incur heavy overheads due to the inherent complexities of the problem, even when targeted for a specific platform. Furthermore, these approaches require that each application fully reveal all tasks and their characteristics so that scheduling can be resolved. Application acceptance or rejection can be accomplished only on a FCFS basis. As described in Section 2.2.5, even PERC's distributed negotiated management of platform resources has a FCFS bias and it is inherently limited in its ability to respond quickly to variations in the program state and available platform resources.

Already mentioned top-layer advantages regarding a single application, apply equally to a dynamic set of concurrent applications. As far as we know, only a top-layer approach provides a model for managing concurrency in a manner which caters to global considerations, i.e. the overall system gain, regardless of the order of arrival. Furthermore, the reduction in complexity enables us to achieve automatic best-effort online adaptation for dynamic sets of concurrent applications on a diverse set of platforms. All of this is accomplished using a *black-box* paradigm in which internal application details are hidden within the jobs of each application, letting each application be developed and maintained independently, freeing the system of the need to collect these details, and allowing vendors to maintain a higher degree of confidentiality. The encapsulated internal scheduler within each job lets each application resolve scheduling independently, choosing the scheduling policies which are most appropriate and effective for its specific set of tasks. Thus, the internal job scheduling policies and overheads will typically be simpler and less expensive than the more elaborate schemes needed to cater to the more general and complex cases, particularly when the task characteristics of future applications are unknown. Diverse applications, e.g. hard, soft and non-real-time applications, can therefore employ completely different scheduling policies within their distinct bandwidths, with the allocation of bandwidths being solely determined by their contribution to system gain.

10.2.6 Reducing Costs

The proposed top-layer approach has several features which should help reduce current high costs of real-time software in the following ways.

Risk Management – As described in Section 1.2, choosing a *cost-effective* run-time platform is a major challenge for complex real-time applications. Conventional approaches essentially force us to commit to a particular platform very early in the system design when precise task requirements and characteristics, e.g. worst-case execution times, can still be quite vague. This commitment incurs high risk. A poor platform selection will either result in exagerated production costs or require more optimized code and a more sophisticated scheduling scheme, thereby incurring higher *development* costs. Even a cost-effective platform choice can become a handicap if platform-related hardware or development tools suddenly become inaccessible, or are found to be faulty. Eventually, even a reliable platform becomes a handicap when portions of it become obsolete. Porting a complex real-time application to a more cost-effective platform, e.g. to a smaller number of more powerful processors, will usually require a re-distribution of tasks among processors as well as new scheduling solutions for each processor. The inherent portability and flexibility of our top-layer approach greatly reduces these risk factors by letting us port to new platforms with little or no extra cost.

Software Reuse and Longevity – Encapsulated jobs serve as real-time objects which can be developed independently and then readily incorporated into programs which require them. This object-oriented design should greatly enhance software reuse and ease software maintenance. The inherent portability of these objects should also greatly increase application longevity by enabling easy porting to new platforms when initial platforms become obsolete.

Quality Assurance – Platform-independent software is a valuable asset even when the targeted platform is indeed appropriate. Application debugging and testing are formidable tasks for many real-time systems, primarily when targeted for specialized operating systems and development environments which generally lack the robust and powerful development tools that are available to developers of more traditional systems. *Portable* real-time applications, on the other hand, can be developed and tested on platforms with the most effective tools available and then ported to the production platforms for final integration when most aspects of the real-time software have been adequately verified.

Software reuse and reduced complexity *also* contribute to the quality and reliability of the application. Software reuse increases object maturity. The lower the complexity, the easier is it to program correctly, test, verify, and understand, the less costly it is to develop and maintain.

10.2.7 Framework Flexibility

As already pointed out by Stankovic and Ramamritham, the conventional layer-by-layer approach is too restrictive. Our top-layer architecture, on the other hand, provides the designer with a very flexible framework for applying the most appropriate method to each part of the system, capable of simultaneous support for a variety of scheduler, meta-controller, and policy-maker policies. When necessary, speculative scheduling policies can be applied to various aspects of the system without jeopardizing the deterministic qualities of other aspects of the system. As demonstrated by ATLAS, top-layer control layers are interchangeable, e.g. we need not use preemptive time-sharing to use online meta-control, so various elements can be adopted independent of the others. The top-layer approach also offers a well-structured framework for reasoning about the various aspects of the system, understanding how its behavior, and analyzing design optimality. We know of no other paradigm which can simultaneously support value-based best-effort performance and deterministic hard real-time scheduling.

Furthermore, conventional design methods are readily incorporated into the top-layer framework, letting us apply different design paradigms to various parts of the system. Top-layer advantages will never be fully realized for subsystems which have only one operational mode, a single internal job state, and are bound to a specific processor. Top-layer overheads can be too significant when task frequencies are high and task execution times are short. Such subsystems are usually best designed using conventional layer-by-layer paradigms, while remaining subsystems can still be designed using a top-layer paradigm. The layer-by-layer subsystems can either be isolated within designated processors, or they can even run at a higher priority on processors serving the portable top-layered subsystems. In the latter case, all we need to do is compute the bandwidth of the layer-by-layer subsystem and deduct it from the processor capacity available to the top-layer jobs. Thus layer-by-layer and top-layer subsystems can readily coexist within any given system. We therefore suggest that subsystems which closely interact with hardware and the outside world, should continue to be designed using a layer-by-layer approach, while higher-level subsystems can gain portability, adaptability, and performance by adopting a top-layer approach.

10.3 Current Top-Layer Limitations

As illustrated by the examples of Section 3.3, and the ATLAS case study of Chapter 8, we strongly believe that the current top-layer models and control architecture can already be applied, as is, to a very significant class of complex real-time applications. Nevertheless, the following inherent and non-inherent limitations must be considered when addressing a specific application.

10.3.1 Inherent Limitations

Inherent top-layer limitations include minimal platform capacities, scheduler-independent jobs, time-sharing overheads, VaPoR overheads, known bandwidths, bandwidth-bound schedulers, clustering fragmentation, mode diversity, optimality dependencies, and timely meta-controller decisions and adaptations.

Minimal Platform Capacities – The most obvious inherent limitation is that platform capacities must at least accommodate the minimal jobs bandwidths for the most demanding job set and job states anticipated. Thus, the total platform capacity must exceed the sum of minimal job bandwidths, and the smallest processor capacity must exceed the largest minimal job bandwidth. Job granularity should therefore be chosen to be small enough to ensure the latter.

Scheduler-Independent Jobs – Jobs in our program model must be *scheduler-independent* in the sense that scheduling in one job can be resolved independently of how scheduling is resolved in other jobs. As described in Section 3.1.2, this does not at all imply that tasks in one job cannot communicate with tasks in another job, but it does mean that inter-job communication must be designed to secure this scheduling independence, e.g. by appropriate asynchronous message passing with adequate queue sizes, as one would do when communicating between processors.

Time-Sharing Overheads – Another inherent limitation is that preemptive time-sharing overheads can become significant if tasks are relatively short, since these overheads must be added twice to each task's overall execution time. As emphasized by Section 4.3 and illustrated by ATLAS, preemptive job time-sharing is not always necessary, and even when necessary, there are several techniques for minimizing these overheads. Nevertheless, there will clearly be cases in which these overheads can be significant enough to require the use of a more powerful platform.

Known Bandwidths – Bandwidth computation for a given set of finite job states will usually be carried out offline, but it can become very nontrivial when the number of possible states is sufficiently large or infinite. An obvious exception to this rule is when Equation (4.2) can be used, e.g. when all job tasks are periodic and independent, and undoubtedly there are additional cases where similar equations can be devised. It is reasonable to assume, however, that there will always be more complex cases where bandwidth computation for jobs with a large number of states will be intractable. For such cases, bandwidths will have to be determined for worst-case states and possibly for an additional limited number of intermediate states.

Bandwidth-Bound Scheduling – Our current job time-sharing schemes assume that actual job bandwidth consumptions will never exceed the allocated bandwidths, i.e. that the assumed internal task execution times are never exceeded. This limitation applies also to most conventional approaches, and it is often resolved by introducing time-out mechanisms which ensure that such time faults are anticipated and treated within the allocated bandwidths.

Clustering Fragmentation – As described in Section 5.1.2, we find that clustering fragmentation can be quite significant when job workloads are predominantly hard and mandatory, if the ratio of jobs to processors is relatively low. For such systems, jobs granularity should be as fine as possible to minimize fragmentation. In most cases, however, hard and mandatory workloads comprise only a fraction of the entire workload. Note, also, that we can always adapt the conventional alternative for hard and mandatory workloads, which is to *staticly* allocate their *worst-case* resource requirements and deduct them from the platform capacities. Thus, job clustering for these workloads is merely an additional alternative to be adopted when useful.

VaPoR Overheads – Another probable source for additional top-layer overheads is the virtual platform for real-time, VaPoR. After all, how can a uniform platform interface exploit the specific architectural aspects of each platform? This might be very true today, but we suggest that this is only a temporary obstacle. Similar arguments were used to delay the use of high-level languages in real-time software. As with high-level languages, we anticipate that advanced operating systems and optimization technologies will eventually provide generic, application-independent, solutions for maximizing the utilization of platform resources without explicit directives from the application.

Mode Diversity – The ability to maximize system gain via adaptive meta-control is limited, of course, by the number of supported job modes and their respective distribution of bandwidths and rewards. The speculative techniques of Chapter 9 are not practical if, unlike ATLAS, agile mode transitions and FIFO-like skippable deadline tolerance is not inborn. Diversified modes and bandwidths, skippable deadlines, and the ability to readily adapt from one mode to another must be primary design objectives.

Optimality Dependencies – When evaluating solution optimality, one must recognize that policy-making optimality depends on meta-control optimality which depends on time-shared optimality. Thus the near-optimality of the top-layer approach is highly dependent on the program decomposition, the frequency of system state transitions, and the required response times.

Timely Meta-Controller Decisions – As implied in Figure 3.4, automated meta-controller decisions must be made at rates which are much less frequent than typical scheduling decisions. Meta-controller decision epochs should be adequately spaced, anyway, to maintain stability and reasonable overheads. More frequent internal state transitions would then have to be accommodated by each job mode within the prescribed bandwidths. This suggests that a *hierarchy* of long-term and short-term modes and meta-controllers might sometimes be necessary.

Timely Meta-Controller Adaptations – Meta-controller decisions must be carried out in a timely manner so as not to disrupt the job activities. When using a cyclic executive to internally schedule job activities, meta-controller adaptations must usually take place at the beginning of each cycle. As demonstrated by ATLAS, relevent adaptations might also be state-dependent. Thus, making decisions and carrying them out must be coordinated in a manner which will guarantee that decisions made can be carried out in a timely manner.

For some applications, the above limitations are crucial. But for a significant number of applications, costs incurred by reduced efficiency can be relatively insignificant when considering other aspects of system development, deployment, and maintenance.

10.3.2 Non-Inherent Limitations

There are, of course, several additional limitations imposed by current control algorithms, many of them *self*-imposed for simplicity sake, which we hope to alleviate with future research. These include independent job modes, scalar bandwidths and gain criteria, additive gain function, no adaptation costs, and scalar platform capacities.

Independent Job Modes – We have thus far assumed, for example, that job mode selection is mutually independent. When non-cyclic, dependencies can be handled in QDP-based meta-control by carrying out the dynamic programming process of Figure 6.15 such that if job j depends on job i, then job j modes are considered only after job i has already been considered and the best job i mode for each dynamic programming state has already been established. G^2 can handle arbitrary dependencies since each iteration already knows what modes have been selected and can accept or reject current candidates accordingly. Complex dependencies will most probably have an adverse effect on meta-control performance in any event, so they should be minimized.

Scalar Bandwidths – Throughout this thesis, job bandwidths, system gain values, and job rewards were all *scalar* values, so our meta-controller algorithms are currently limited to computational-intensive applications and shared-memory multi-processor platforms. When using a global reflective memory network, our current meta-control algorithms can even be applied to *distributed* applications, since I/O overheads are fixed and worst-case inter-job traffic is accommodated regardless of how jobs are load-balanced over the processors. For the more general case, however, we must also manage memory and communication resources, while catering to varying communication throughputs between processors, distributed notions of time, and more. We believe that this is readily accomplished by appropriate extensions to our current meta-controller algorithms, e.g. by converting the bandwidth, reward, and gain scalars into vectors. Bandwidth *vectors* would serve to represent additional resource requirements, e.g. I/O and memory. Note, also, that the algorithms of Chapter 4 are just as appropriate for I/O time-sharing as they are for computational time-sharing, so they can serve for the timely transfer of I/O as well. For shared memory platforms, job load-balancing complexities should still be similar to those required by scalar bandwidths. A brute-force extension of QDP, for example, would simply require a *vector* of quantum values to determine the granularity of each coordinate in the dynamic programming state vector. The number of dynamic program states would grow significantly, but this could be restrained by using a finer quantum for those resources which are more critical. A possible G^2 extension would be to use a weighted sum of gradients for each of the vector coordinates, weights being determined by resource criticality. Note that implications to overall G^2 time complexity is much less pronounced than in QDP.

Scalar Gain Criteria – Scalar gain values are also not always adequate. In the ATLAS testbed, for example, one might want to distinguish between response time and SNR performance, with the former being constrained to never exceed a given value and the latter to be maximized. Here again, we contend that the above algorithms can be readily extended to accommodate multi-variable gains and rewards, since only one of the coordinates must be maximized and all other coordinates need only be constrained.

Additive Gain Functions – The meta-controller solutions devised in this thesis have all assumed an additive gain function, but this is not an inherent limitation. Best-effort scheduling schemes generally assume additive value functions, and an additive gain function served ATLAS quite adequately. Nevertheless, the examples in Section 3.3 indicate that there are a great many applications which would require gain functions which are not additive. Appropriate meta-controller algorithms must therefore be devised to cater to a wider range of gain functions. The primary QDP restriction is that v^* be definable recursively so that we can apply dynamic programming. The notion of a gradient used by G^2 appears to be even more accommodating. We therefore believe that current algorithms are readily adapted to support a richer family of gain functions, e.g. a product gain function, but we have yet to evaluate their performance for such functions.

No Adaptation Costs – As in all existing alternatives, job migration and mode transition costs are currently overlooked. Only the job context has to be migrated if code for all jobs is maintained on each processor, and this is indeed the case on shared memory platforms and in distributed systems with global reflective memories. Mode transition costs generally *can*, indeed, be overlooked when merely altering rates, resolutions, or scheduling policies. When significant, transition costs might be accounted for by an appropriate penalty within the reward function.

Scalar Platform Capacities – The scalar capacity characterizations of Section 3.1.1 produce pessimistic capacity assumptions when there are non-linear differences in the platform execution times within a given platform family. We believe, however, that a vectorized capacity characterization should better accommodate such differences, and also differentiate between the average and worst-case capacities to better accommodate hard and soft workloads.

10.4 Future Extensions

As already mentioned, we believe that our top-layer approach can readily be extended to cater to distributed, fault-tolerant, real-time applications with even greater scheduling and meta-control complexities. To illustrate how this might be one, we show how our approach might serve as the basis for solving an exciting new problem which has yet to addressed: the management of dynamic sets of fault-tolerant real-time software agents on a network.

Community-Minded Real-Time Software Agents

The term *software agents* is often used to describe relatively small, adaptive, distributed, hopefully mobile, continuously-running, self-reliant pieces of code capable of cooperating with other such agents over a network to obtain goal-based behavior in a very dynamic and uncertain environment. Current research is primarily concerned with how to develop various agent-based applications - an *agent-based application* being one that requires one or more software agents. Our concern, however, is on how to *manage* a large dynamic set of such applications over a dynamic network configuration so that unrelated applications can co-exist without interfering with each other. This issue of resource management for adaptive software agents has yet to be addressed, and it becomes more and more crucial as the number of available agent-based applications outgrow available network capacities. We also require that this resource management fully accommodate the needs of concurrent *real-time software agents*, i.e. agents with complex real-time requirements.

Consider, for example, a heterogeneous network with two classes of agent-based applications: (a) a set of real-time control applications which control and monitor robots on a factory floor, and (b) a set of autonomous agent applications which collect and analyze factory data to seek ways to improve the factory's overall performance. All robots are connected to the factory network which serves as a shared platform for agents in both sets of applications. Different sets of robots serve different phases of factory production. Autonomous agent applications respond differently to different factory states, the complexity of their decision-making processes is highly data-dependent, and their decisions affect the factory production process. Consequently, the set of concurrent agents and their required resources is very dynamic. We also assume that the sum of required agent resources often exceeds the available network resources, so resource management must be *community-minded*, i.e. cooperative in a manner which considers the needs of both the robots and the factory owners.

Distributed Matrix Management

Each independent agent-based application clearly requires some form of application management within the application, whether fixed and hierarchical as in some robot architectures, or whether dynamic and distributed as in many voting and bidding schemes. To support *community-minded* management, we require that each application have any number of selectable work point (WP) alternatives, each WP requiring a different agent task force with well-defined resource requirements, whereupon the application will contribute an appropriate vector of global gain measures when operat-

Control Layer	Level	Function	Typ. Rate
Application Policy-Making	6	when, where (scope) & how (method & depth)	several minutes
Application Meta-Control	5	WP selection & global load balancing	several minutes
Job Policy-Making	4	when, where (scope) & how (method & depth)	per minute
Job Meta-Control	3	mode selection & local migration	per second
Real-Time Time-Sharing	2	job time-sharing	msecs
	1	internal job scheduling	

Figure 10.1: Distributed Top-Layer Software Control

ing at that WP. We further assume that the set of applications, their WPs and the WP attributes are all state-dependent and therefore very dynamic. The network configuration and available network resources may also vary with time, so resource management and application management are clearly continuous and reactive, thereby qualifying as agent-based applications in their own right.

Network resource management must cooperate with application management at two levels. At the higher level, the resource management must somehow consider total WP requirements and contributions of all active applications so that it can choose a WP for each application which will be most beneficial, overall, to some global notion of system gain without overloading the network. At the lower level, each active application agent must then be provided with the essential resources corresponding to the agreed WP. The choice of application WPs must be fully coordinated with the application management because it dictates which set of agents will be active, whereupon agent resource allocation can be carried out at the independent agent level.

A Distributed Top-Layer Architecture

Borrowing from our top-layer architecture, we suggest that each agent be implemented as a state-dependent job with several mode alternatives, but we assume that it would be impractical to reevaluate the application WPs with each agent state-transition. We therefore propose adding additional control layers to our current control architecture, as depicted in Figure 10.1. The functions of the bottom four layers remain the same, but the job meta-control algorithms will probably be superceded by the distributed decision-making technologies being developed for collaborating software agents. Note, also, that the additive joint schedulability test of our job time-sharing architecture should allow us to carry out centralized meta-control decisions over a network even without *bidding* or *focussed addressing* (see Section 2.2.2).

The functions of the top two layers in the extended control architecture should carry out policy-making and meta-control at the application level. This produces two degrees of meta-controller freedom which are carried out at two hierarchical levels. At the higher level, application WPs are typically reevaluated every few minutes, thereby dictating which jobs are active, roughly limiting total resource bandwidths for each application, and possibly determining initial job modes corresponding to the selected WPs. Finer job mode selection and resource management is then carried out more frequently at the lower job level to best-effort adapt to internal application state-transitions, using methods similar to those in this thesis. For quicker response times, job meta-control might also be internally layered, with local job meta-controllers affecting immediate local mode-transitions to best-accommodate state-transitions until higher gain adaptations are determined by global meta-controllers and propagated to all processors on the network. Major policy-making issues would then be to manage these local and global meta-controller activities in a cost-effective manner. The top-layer framework is thus readily extended to support distributed systems, as well.

Fault Tolerance

The challenge of fault-tolerant systems is to manage robustness in a manner which minimizes the probability of system failure for all anticipated hardware and software faults. Job rewards for redundant jobs should therefore reflect the contribution of that job to a reduced MTBF.

Redundancy management can be incorporated into the software control model in any of several ways. One possibility would be to predetermine a maximum total capacity, β_n, which is to be allocated to non-redundant jobs, activate the meta-controller regarding those jobs only, and then use a DG algorithm to allocate remaining capacity to those redundant jobs which will maximize the MTBF. For a more evenly spread distribution of redundancies, we might use a modified BFD clustering policy in which a best fit is when the subsequent processor utilization is closest to $\beta_c \cdot \beta_n / \beta$.

Fault-tolerance is thus improved in two ways. Firstly, the top-layer application is inherently fault-tolerant in the sense that it can automatically best-adapt to hardware faults by removing the faulty hardware capacities from the list of available capacities and activating the online meta-controller. Secondly, we can frequently reevaluate redundancy management using the above methods to best accommodate each new platform state in light of the latest fault-related statistics accumulated during run-time.

10.5 Food For Thought

In classical computer science, maximizing processor utilization is a very common criteria for evaluating a software control method. Halang and Stoyenko contend, however, that for *embedded* real-time systems, optimal utilization is irrelevant [HS91]. Costs have to be seen in a larger context. When the total cost of a failed system is considered, the cost of any processor involved is usually negligible. Low processor utilization is a cheap price to be paid for *simplicity* as a prerequisite to achieve dependability and predictability.

Similar sentiments were expressed by David Parnas in a keynote address at the inaugural IEEE International Conference on Engineering of Complex Computer Systems, ICECCS. True, the world has inherent complexity. We must deal with complex devices, must support humans who have complex tasks, and must interact with previously built systems. But *complex* computer systems for their own sake are not our goal. Parnas does not want to be known as one who has built several large complex computer systems, but rather as one who has built *simple* systems that perform complex tasks.

Promoters of the layer-by-layer approach posit that complex requirements can and should be met by *simple* implementations. Real-time scheduling should never become a problem, so when multitasking requirements get complicated, simply divide and conquer by adding additional inexpensive processors to the system. Borrowing from our top-layer terminology, each job should be isolated in a dedicated processor and job task characteristics should be generally very simple. Internal job scheduling should be predominantly sequential with very limited levels of interrupt-driven activity. To maintain simplicity, communicating jobs should use very simple protocols. When several modes are supported, simply solve independently for each mode or add additional hardware to support each mode. This quest for simplicity has already paid off handsomely in many automated industries and other areas.

Unfortunately, layer-by-layer simplicity is not always attainable. Borrowing from a quotation attributed to Albert Einstein, "Everything should be as simple as possible, *but not simpler*." The quest for simplicity fails when hardware expense becomes critical or when the number of possible mode combinations grow exponentially. In the age of global networking, a growing proportion of real-time activities will no longer be embedded, whereupon platform costs will no longer be negligible. As illustrated by the multimedia example of Section 3.3.1, there is a growing market for powerful multi-purpose systems, fueled by the exponential growth in low-cost capacities, to which simple scheduling paradigms cannot be applied. Computational platforms are usually bought with very limited *a priori* knowledge regarding the applications it will host. The consumer wants the system to grow with his needs, and it is unlikely that simple hardware add-ons for every new function will be an acceptable solution. A pilot flying a plane and a skipper steering his ship cannot possibly anticipate all of their needs before embarking on a mission. However, when demanding circumstances arise, they want to get as much as possible from the computational platforms they have available. Such systems require the complex and concurrent real-time requirements addressed by this thesis.

The recent establishment of the IEEE ICECCS framework, is ample proof that solutions for complex real-time systems are vital to a widening range of applications. The ICECCS call for papers talks of complex computer systems in transportation, manufacturing, communications, defense, hazardous environments, aerospace, energy, health care and other sectors. Special areas of interest include:

- Methods for reducing complexity at the specification, design and implementation stages by considering the structure of the system (e.g. decomposable design methods, restricting resource contention at the scheduling level, etc.)
- Methods for reducing complexity, by intelligently fusing and converting information into cost and objective functions.
- Heuristics for the general allocation problem that may be scaled up to complex and very large systems.
- Approaches to facilitate evolution in systems (as found due to changes, re-use, adaptability, etc.).

- Tradeoffs in predicting satisfiability of various criteria (e.g. faster performance estimation versus more accurate projections).
- Realistic case studies and practitioner experience reports.

We suggest that conventional approaches fail because they tend to focus on very specific aspects of a system, often seeking elusive *optimal* solutions to only part of the problem. Consequently, these frameworks are rarely adopted by the industrial community because they end up being too expensive, overly restrictive, and therefore impractical.

The top-layer approach described in this thesis is a holistic approach which seeks a balance, and each of the above areas of interest has been addressed, somewhat, by this thesis. Rather than seek the optimal, we settle for *near*-optimal potential and seek ways to exploit this potential. We provide a simple and modular program model which enables us to reason about the various aspects of the system using simple additive rules. Rather than focus on one criterion of optimality, we seek a balance between several important criteria. Even when time-sharing overheads are relatively high, thereby requiring a more powerful platform, advantages in all other realms might make the approach very worth while. Consequently, while we deem the approach to be *essential* to the complex real-time applications of our problem domain, we suggest that it may still be very valuable for simpler applications as well.

A major issue in classical computer science is to establish relevant criteria for evaluating and comparing solutions. As we have already shown, the top-layer approach has significant advantages when it comes to engineering complexity, scalability, performance, portability, concurrency, reduced costs, and the flexible nature of the design framework. We have also found that the approach has inherent limitations and that the current solutions presented by this thesis have additional limitations which should disappear with time. Clearly there will be systems, e.g. many embedded systems, which will find the top-layer approach unnecessary and too costly. So which approach is better?

When analyzing the underlying philosophies of each approach, we find that the two approaches are not actually diametrically opposed. Both approaches stress the issue of predictability, but whereas the top-layer approach focuses on portability and performance, the layer-by-layer approach stresses simplicity. Thus, they really differ only in the importance that they attribute to various aspects of system design, and many elements of each approach can actually be adopted by the other. As already pointed out, the RtTS architecture is readily incorporated into a layer-by-layer context, just as layer-by-layer methods are readily applied to arbitrary modes in a top-layer architecture. This brings us to suggest that next generation systems might be designed using a *hybrid* approach in which each subsystem will be designed using an appropriate balance between paradigms.

10.6 Summary

The top-layer paradigm described in this thesis is a holistic approach which attempts to simultaneously address complexity, scalability, performance, portability, adaptability, concurrency, an reuse, all in one stroke. We know of no other paradigm which can provide low-cost best-effort performance for dynamic sets of hard real-time systems with complex real-time characteristics on a diverse set of platforms. To establish the potential in this approach, we have tackled several of the more crucial, ground-breaking, aspects:

- the development of appropriate *job-oriented* program and platform *models*,
- a *layered* software control model for hard job time-sharing and best-effort, online, load-balancing, meta-control and policy-making,
- effective *algorithms* for each software control layer,
- a comprehensive *portable* software architecture,
- a *black-box* job paradigm for *concurrent* applications,
- a realistically complex top-layer ATLAS *case study* to prove feasibility,
- using the case study to demonstrate how a top-layer approach might *better accommodate non-determinism* within the application and the run-time platforms and
- evaluating the paradigm strengths and weaknesses.

We have also listed examples for which it clearly has advantages, and shown why we believe that it can be extended to benefit a much wider range of complex systems, e.g. distributed, fault-tolerant, real-time systems.

It is only a beginning. As new paradigms often do, the proposed architecture has opened up several *new* avenues for further study, e.g.

- optimal application decomposition into jobs,
- minimized time-sharing overheads,
- improved bandwidth computation methods,
- automated job scheduler selection,
- potent families of system gain functions,
- generic meta-control algorithms for vectorized bandwidths and rewards,
- accommodating job migration and mode-transition costs,
- hierarchical meta-controller schemes,
- experimenting with preemptive multiprocessor platforms,
- platform noise and task profiling,
- the efficient accommodation of transient overloads due to sub-WCET assumptions,
- effective VaPoR-like abstractions,
- and more.

Bibliography

[ATY87] A. K. Agrawala, S. K. Tripathi, and Yuan. Scheduling in real-time distributed systems - a review. Technical report, Department of Computer Science, University of Maryland, December 1987. TR 1955.

[Bak91] T. Baker. Stack based scheduling of real-time processing. *Journal of Real-Time Systems*, 3(1):67–100, March 1991.

[BF81] Benjamin S. Blanchard and Wolter J. Fabrycky. *Systems Engineering and Analysis*. Prentice-Hall Inc., 1981.

[BGW91] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System - Load Balancing for Unix*. Number 672 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1991.

[BN94] Swagato Basumallick and Kelvin Nilsen. Incorporating caches in real-time systems. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, April 1994.

[Bok79] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Trans. on Software Engineering*, SE-5(4):341–349, July 1979.

[BS88a] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and Ada. In *Proceedings Ninth Real-Time Systems Symposium*, pages 120–129, 1988.

[BS88b] Thomas Bihari and Karsten Schwan. A comparison of four adaptation algorithms for increasing the reliability of real-time software. In *Proceedings Ninth Real-Time Systems Symposium*, pages 232–243, March 1988.

[BS91] Thomas Bihari and Karsten Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems*, 9(2):143–174, May 1991.

[Car84] G. D. Carlow. Architecture of the space shuttle primary avionics software system. *Communications of the ACM*, 27:926–936, September 1984.

[CB93] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, December 1993.

[CDHC94] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. Deterministic scheduling for real-time integrated avionics. In *Proceedings 12th Real-Time Systems Symposium*, pages 165–169, December 1994.

[CL90] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real time systems. *Journal of Real-Time Systems*, 2(4):325–346, November 1990.

[Der74] M. L. Dertouzos. Control robotics: The procedural control of physical processes. In *Proc. IFIP Congress*, pages 807–813, 1974.

[DL78] S. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[DM89] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard real-time tasks. *IEEE Trans. on Software Engineering*, 15(12):1497–1506, December 1989.

[DRSK89] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz. The real-time operating system of MARS. *ACM Operating Systems Review*, 23(3):141–157, July 1989.

[FGG+91] B. Furht, D. Grostick, D. Gluch, G. Rabbat, J. Parker, and M. Roberts. *Real-Time Unix Systems.* Kluwer, 1991.

[FR73] B. L. Fox and J. E. Rolph. Adaptive policies for markov renewal programs. *Annals of Statistics*, 1:334–341, 1973.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability - Guide to the Theory of NP-Completeness.* Bell Telephone Laboratories, Inc., 1979.

[GS89] Prabha Gopinath and Karsten Schwan. CHAOS: why one cannot have only an operating system for real-time applications. *ACM Operating Systems Review*, 23(3):106–125, July 1989.

[GS90] Ahmed Gheith and Karsten Schwan. Chaosarc: Kernel support for multi-weight objects, invocations, and atomicity in real-time applications. Technical report, GIT-ICS-90/06, College of Computing, Georgia Institute of Technology, January 1990.

[HS89] Dieter Haban and Kang G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. In *Proceedings Ninth Real-Time Systems Symposium*, pages 172–181, December 1989.

[HS90] Wolfgang A. Halang and Alexander D. Stoyenko. Comparative evaluation of high-level real-time programming languages. *Journal of Real-Time Systems*, 2(4):365–382, November 1990.

[HS91] W. A. Halang and A. D. Stoyenko. *Constructing Predictable RT Systems.* Kluwer Academic Publishers, 1991.

[Hsu90] Liangchuan Hsu. Multiprocessor scheduling for hard real-time software. Technical report, Naval Postgraduate School, Monterey, California, June 1990. Masters Thesis.

[IK75] Oscar H. Ibarra and Chul E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975.

[ITM92] Yutaka Ishikawa, Hideyuki Tokuda, and Clifford W. Mercer. An object-oriented real-time programming language. *IEEE Computer*, 25(10):66–72, October 1992.

[JDU+74] D. S. Johnson, A. Demers, J. D. Ullmann, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one dimensional packing algorithms. *SIAM Journal of Computing*, 3:299–325, 1974.

[KL91] Kevin B. Kenny and Kwei-Jay Lin. Building flexible real-time systems using the Flex language. *IEEE Computer*, 24(5):70–78, May 1991.

[KM85] J. Kramer and J. MaGee. Dynamic configuration for distributed systems. *IEEE Trans. on Software Engineering*, 11(4):424–436, April 1985.

[KRP+93] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.

[KS95] Gilad Koren and Dennis Shasha. The anti-task algorithm: An approach to handling overloaded systems that allow skips. In *Proceedings 13th Real-Time Systems Symposium*, December 1995.

[Lap93] Phillip A. Laplante. *Real-Time Systems Design and Analysis*. IEEE Computer Society Press, 1993.

[Law77] E. L. Lawler. Fast approximation algorithms for knapsack problems. In *18th Annual Symposium on Foundations of Computer Science*, pages 206–213, October 1977.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, 20(1):46–61, January 1973.

[LL94] Jyh-Charn Liu and Hung-Ju Lee. On the worst-case execution time of cached real-time programs. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, April 1994.

[LLS+91] Jane W. Liu, Kwei-Jay Lin, Wei-Kuan Shih, Albert Chuang shi Yu, Jen-Yao Chung, and Wei Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.

[LML+94] Sung-Soo Lim, Sang Lyul Min, Minsuk Lee, Chang Yun Park, Heonshik Shin, and Chong Sang Kim. Portable execution time analysis for risc processors. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, April 1994.

[Loc86] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie-Mellon University, Dept. of Computer Science, Pittsburg, Pa. 15213, 1986.

[Loc92] C. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Journal of Real-Time Systems*, 4(1):37–54, 1992.

[LSD89] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[LSS87] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environments. In *Proceedings of 8th Real-Time Systems Symposium*, pages 261–270, December 1987.

[LTCA89] Shem-Tov Levi, Satish K. Tripathi, Scott D. Carson, and Ashok K. Agrawala. The MARUTI hard real-time operating system. *ACM Operating Systems Review*, 23(3):90–105, July 1989.

[McB94] Oliver A. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–444, 1994.

[McE94] Charlie McElhone. Adapting and evaluating algorithms for dynamic schedulability testing. Technical report, Department of Computer Science, University of York, England, February 1994.

[MD78] A. K. Mok and M. L. Dertouzos. Multiprocessor scheduling in a hard real-time environment. In *Proc. of the Seventh Texas Conference on Computing Systems*, November 1978.

[Mil90] F. W. Miller. Predictive dealine multi-processing. *ACM Operating Systems Review*, 24(4):52–62, October 1990.

[MJ70] R. R. Muntz and E. G. Coffman Jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of ACM*, 17(2):324–38, April 1970.

[Mok84] A. K. Mok. The design of real-time programming systems based on process models. In *Proceedings IEEE 1984 Real-Time Systems Symposium*, pages 5–17, December 1984.

[MS92] Bodhisattwa Mukherjee and Karsten Schwan. A survey of multiprocessor operating system kernels. Technical report, GIT-CC-92/05, College of Computing, Georgia Institute of Technology, January 1992.

[MW94] Sharad Malik and Andrew Wolfe. Harnassing the performance of microprocessors for real-time applications. In *Proceedings of the Workshop on Architectures for Real-Time Applications*, April 1994.

[Nil95] Kelvin Nilsen. Issues in the design and implementation of real-time java. Technical report, November 1995.

[Oh94] Yingfeng Oh. *The Design and Analysis of Scheduling Algorithms for Real-Time and Fault-Tolerant Computer Systems*. PhD thesis, University of Virginia, May 1994.

[OSS90] Dave Ogle, Karsten Schwan, and Richard Snodgrass. The dynamic monitoring of real-time distributed and parallel systems. Technical report, GIT-ICS-90/23, College of Computing, Georgia Institute of Technology, May 1990.

[Pap65] Athanasios Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw-Hill Kogakusha, Ltd., 1965.

[Ram91] K. Ramamritham. Real-time concurrent c: A language for programming dynamic real-time systems. *Journal of Real-Time Systems*, 3:337–405, 1991.

[Rea86] James F. Ready. VRTX: A real-time operating system for embedded microprocessor applications. *IEEE Micro*, 6(4):8–17, August 1986.

[RS84] K. Ramamritham and J. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, July 1984.

[RS91] K. Ramamritham and J. A. Stankovic. Scheduling strategies adopted in spring. Technical report, University of Massachusetts, 1991.

[RSS90] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[Sah75] Sartaj Sahni. Approximatie algorithms for the 0/1 knapsack problem. *Journal of the ACM*, 22(1):115–124, January 1975.

[SBWT87] Karsten Schwan, Thomas Bihari, B. Weide, and G. Taulbee. High-performance operating system primitives for robotics and real-time control systems. *ACM Transactions on Computer Systems*, 5:189–231, August 1987.

[SC88] John A. Stankovic and Sheng-Chang Cheng. *Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey*, pages 150–173. IEEE Computer Society Press, 1988.

[SGZ90] Karsten Schwan, Ahmed Gheith, and Hongyi Zhou. From chaos-min to chaos-arc: A family of real-time kernels. In *Proceedings of the Real-Time Systems Symposium, Orlando, Florida*, pages 82–92, December 1990.

[SR89] John A. Stankovic and Krithi Ramamritham. The Spring kernel: A new paradigm for real-time operating systems. *ACM Operating Systems Review*, 23(3):54–71, July 1989.

[SR90] John A. Stankovic and Krithi Ramamritham. Editorial: What is predictability for real-time systems? *Journal of Real-Time Systems*, 2(4):247–254, November 1990.

[SRL90] Lui Sha, Rugunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[SSL89] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

[SSNB95] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, June 1995.

[Sta93] John A. Stankovic. Resource allocation in real-time systems. *Journal of Real-Time Systems*, 5(2/3), May 1993. Editorial.

[Sto87] Alexander D. Stoyenko. A schedulability analyzer for Real-Time Euclid. In *Proceedings Eighth Real-Time Systems Symposium*, pages 218–227, 1987.

[SZ92] Karsten Schwan and Hongyi Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Trans. on Software Engineering*, 18(8):736–748, August 1992.

[TM89] Hideyuki Tokuda and Clifford W. Mercer. ARTS: A distributed real-time kernel. *ACM Operating Systems Review*, 23(3):29–53, July 1989.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-time Mach: Towards a predictable real-time system. In *Proceedings of USENIX 1990 Mach Workshop*, pages 73–82, October 1990.

[WAB+89] Horst F. Wedde, Ghasem S. Alijani, Willi G. Brown, Shengdong Chen, Gookhai Kang, and Bo-Kyung Kim. Operating system support for adaptive distributed real-time systems in DRAGON SLAYER. *ACM Operating Systems Review*, 23(3):126–140, July 1989.

[Whi92] D. J. White. *Markov Decision Processes*. John Wiley & Sons, 1992.

[XP93] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Trans. on Software Engineering*, 19(1):70–84, January 1993.

[YS91] Michal Young and Lih-Chyun Shu. Hybrid online/offline scheduling for hard real-time systems. Technical report, SERC-TR-100-P, Software Engineering Research Center, Department of Computer Sciences, Purdue University, May 1991.

[ZRS87] Wei Zhao, Krithi Ramamritham, and John A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(8):949–960, August 1987.

Appendix A

ATLAS Requirements

ATLAS is a portable software application which synthesizes polyphonic MIDI-controlled music in real-time, for an arbitrary set of musical instruments. As depicted in Figure 1.2, ATLAS configuration inputs include **wave-form tables** and **synthesis methods** libraries to customize the list of supported instruments and the sets of alternate synthesis modes for each instrument. The versatile ATLAS configuration scheme is described in Section A.1.

Dynamic online inputs include the stream of MIDI events which prescribe the notes which ATLAS must synthesize for each instrument, as well as an interactive GUI (graphic user interface) for activating available synthesis and display options. ATLAS outputs include a buffered audio output, i.e. the FIFO in Figure 1.2, a graphic display of the notes being played and the audio output volume, and a recorded trace of all internal ATLAS activities so that ATLAS timeliness and gain performance can be offline verified and evaluated. The MIDI interface is discussed in Section A.2, the synthesized audio output in Section A.3, the interactive GUI in Section A.4, online trace recording in Section A.5, and targeted ATLAS platforms in Section A.6.

A.1 Versatile Configuration

The primary ATLAS function is to digitally emulate a polyphonic MIDI-controlled electronic synthesizer in software, i.e. to play music for up to 16 musicians playing an arbitrary set of configurable instruments. Using standard MIDI terminology, each musician is referred to as a *channel*, and each channel has a *program* which defines the nature of the sound to be produced by that musician, whether a conventional instrument, e.g. a piano, folk gui-

```
SampleRate=8000;       % max rate in hertz
TimeFrame=0.1;         % reflecting beat resolution in seconds
Polyphony=3;           % max concurrent notes
ProgramLibrary=atlas.plb;   % synthesis methods
Program 0 piano;       % instrument for \Midi{} program 0
Program 1 harpsichord;% instrument for \Midi{} program 1
Program 2 organ;       % instrument for \Midi{} program 2
Program 3 accordion;   % instrument for \Midi{} program 3
Program 4 guitar;      % instrument for \Midi{} program 4
Program 5 violin;      % instrument for \Midi{} program 5
WaveformLibrary=atlas.wlb;  % oscillator and ADSR waveforms
MidiFile=atlas.mid;    % when using \Midi{} emulation
Profile=atlas.pro;     % assumed timing characteristics
```

Figure A.1: A Typical ATLAS Configuration File

tar, violin, trumpet, flute, or drum, or something more exotic, e.g. a whistle, a scratch, or a human. Channels may dynamically switch programs at any time in between the notes being played by them. Playing a chord in any given channel requires generating several notes at the same time. Different channels play their instruments at different times. Thus the number and nature of concurrent notes is quite uncertain and very dynamic.

As illustrated in Figure A.1, a textual *configuration* file is used to define available program numbers and to assign an instrument to each program number. Program 0 is the default program for all MIDI channels, and in this example it has been configured to represent a piano. Such a mapping of program numbers to instruments is typically carried out by a MIDI *mapping* utility in any MIDI-supporting system. The SampleRate variable in the configuration file determines the sample rate, f_s, for the mixed audio output depicted in Figure 1.2. The TimeFrame variable determines the MIDI timing resolution, i.e. MIDI notes will always begin and end at time multiples of TimeFrame. Consequently, ATLAS requires that best-effort decisions as to which synthesis methods should serve to generate each note will also be carried out at the designated TimeFrame rate. The Polyphony variable limits the number of concurrent notes, so that adequate worst-case memory resources can be pre-allocated.

Following these parameters, this primary configuration file contains several variables which identify additional configuration files. The Program-Library variable identifies a file containing alternate synthesis methods for each program, and the WaveformLibrary variable identifies a file containing the waveform tables serving these synthesis methods. When wishing to carry out repetitive ATLAS experiments, or when simply lacking a MIDI keyboard and musician, ATLAS supports using a recorded MIDI file to emulate the online MIDI input, whereupon the MidiFile variable indicates which MIDI file is to be used. When running ATLAS on a familiar runtime platform, one may also specify a Profile file containing known execution time statistics for each of the configured synthesis methods and all other ATLAS activities so that appropriate computational resources can be determined.

A.1.1 A Program Library

Figure A.2 contains a typical instrument specification in a program library file. In this example, we have four synthesis methods to choose from when playing a piano. Referring back to Figure 1.3, we readily recognize that Add2 and Add3 are *additive* synthesis methods. Each method specification begins with a symbolic id, e.g. Add3, followed by a list of interconnected synthesis nodes, e.g. oscillators (osc), envelope generators (env), and mixers (mix). The Add3 synthesis method, for example, is displayed in the upper left corner of Figure A.3. Here, we can see its five synthesis nodes, their symbolic ids, and their interconnections. All program libraries have a file name extension of plb, and they can contain an arbitrary number of instruments, methods, and nodes.

A.1.2 A Synthesis Method

As shown in Figure A.2, each method has a designated output node, e.g. output=mix, as well as a pair of reward values. The latter reward reflects that method's SNR when generating samples at a maximum f_s sample rate. The former reward reflects the SNR when generating at the minimal Nyquist sample rate, i.e., the minimal sample rate which is at least twice that of the highest spectral component to be produced by that method[1]. We elaborate a bit more on these two rewards in Section A.3.1.

Each method node has a symbolic id, x,y coordinates controlling how it is displayed, as well as all the synthesis parameters required by each

[1]Computing precise SNR values for each synthesis mode is beyond the scope of this research, so current files contain ad hoc values to illustrate how it would work.

```
Program id=Piano;
Mode id=Add3 output=mix reward=(0.4,0.9)
{
 x=0.2 y=0.5
 Env id=env Adsr=Linear volume=Midi Attack=Midi Release=Midi Sustain=10 Hz;
 x=0.5 y=0.2
 Osc id=Osc1 wave=Sine am=env pitch=Midi;
 x=0.5 y=0.5
 Osc id=Osc2 wave=Sine am=env pitch=Midi volume=0.2 transpose=12;
 x=0.5 y=0.8
 Osc id=Osc3 wave=Sine am=env pitch=Midi volume=0.1 transpose=36;
 x=0.8 y=0.5
 Mix id=mix  signal={osc1,osc2,osc3};
}
Mode id=Add2 output=mix reward=(0.4,0.8)
{
 x=0.2 y=0.5
 Env id=env Adsr=Linear volume=Midi Attack=Midi Release=Midi Sustain=10 Hz;
 x=0.5 y=0.3
 Osc id=Osc1 wave=Sine am=env pitch=Midi;
 x=0.5 y=0.7
 Osc id=Osc2 wave=Sine am=env pitch=Midi volume=0.2 transpose=12;
 x=0.8 y=0.5
 Mix id=mix  signal={osc1,osc2};
}
Mode id=Wave1 output=osc reward=(0.3,0.7)
{
 x=0.3 y=0.5
 Env id=env Adsr=Linear volume=Midi Attack=Midi Release=Midi Sustain=10;
 x=0.7 y=0.5
 Osc id=osc wave=Wave1 am=env pitch=Midi;
}
Mode id=Sine output=osc reward=(0.2,0.5)
{
 x=0.3 y=0.5
 Env id=env Adsr=Linear volume=Midi Attack=Midi Release=Midi Sustain=10;
 x=0.7 y=0.5
 Osc id=osc wave=Sine am=env pitch=Midi;
}
```

Figure A.2: A Typical Program Library

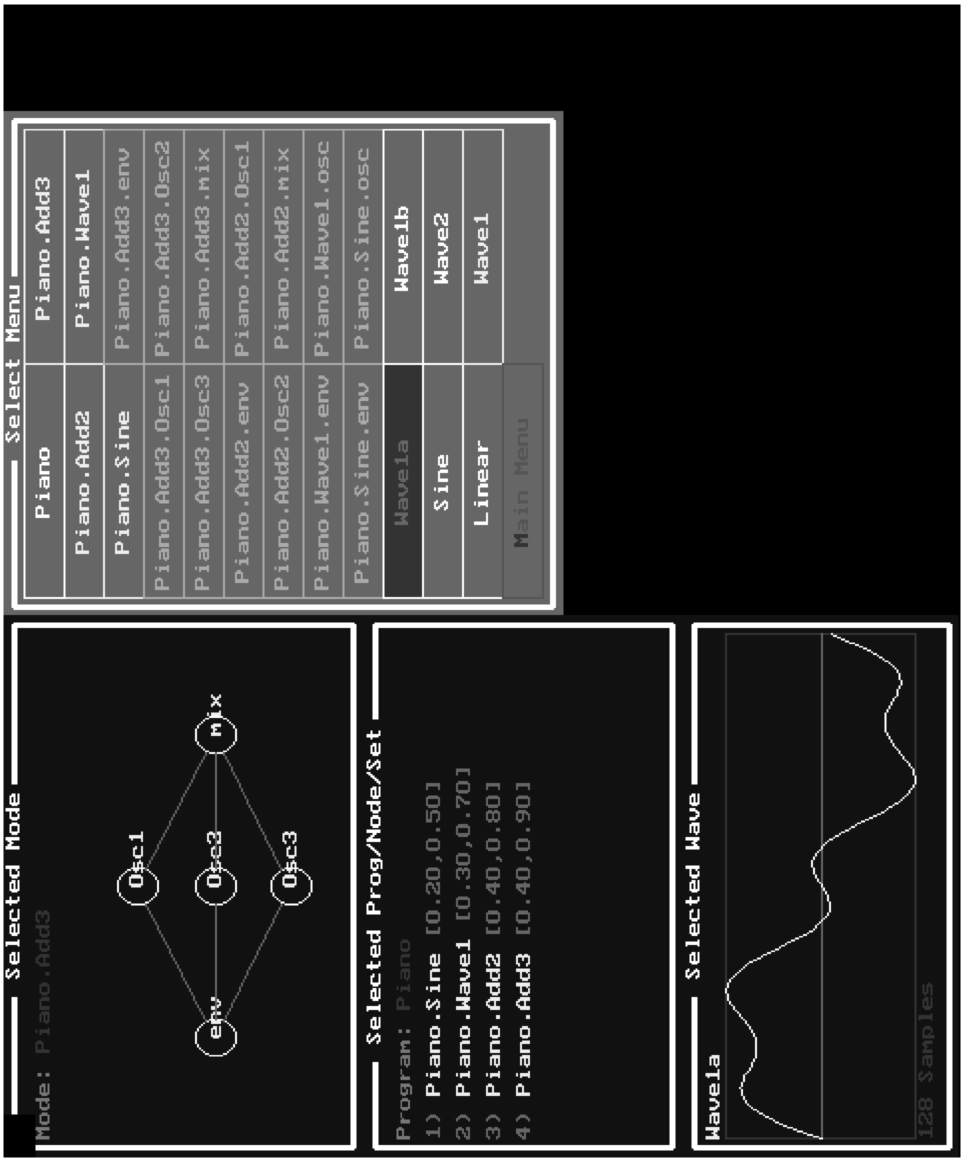

Figure A.3: Displayed Program Library

node type. Thus the osc2 oscillator node in mode add3 of the instrument *piano* is readily identified as piano.add3.osc2, and the oscillator synthesis parameters include the id of an oscillator waveform (wave=sine), the id of an amplitude modulator node (am=env), an oscillation pitch (pitch=midi), a volume (volume=0.2), and an optional transpose (transpose=12) indicating how many half tones should be added to the pitch. Thus, the pitch for piano.add3.osc2, in this example, will be one octave (12 half tones) higher than the pitch dictated by the current midi input.

A.1.3 Attack, Sustain, and Release

Pressing an instrument key initiates a note *attack*, wherein the amplitude of the emitted sound has to rise to a given level. Typical attacks will overshoot and decay, so the attack phase is often subdivided into attack and *decay* segments. Releasing a key will initiate a note *release*, whereupon the amplitude of the emitted sound must descend to zero. The *velocity* of the attack and the release will usually be determined by the state of a level or by the way the velocity-sensitive or pressure-sensitive key is being pressed or released. Between the attack and release phases, the emitted amplitude must be *sustained*, and the sustained amplitude might be variable or fixed. The duration of this sustained phase is arbitrary so the sustained amplitude is usually described as a periodic waveform. The dynamic amplitude levels throughout the attack, sustain, and release phases are collectively referred to as an ADSR (attack-decay-sustain-release) envelope.

The envelope generator, piano.add3.env, in Figure A.2, must therefore be assigned the id of an ADSR envelope (adsr=linear) which described the envelope waveform, a volume (volume=midi) to determine the envelope scale, an attack velocity (attack=midi), an release velocity (release=midi), and a sustain frequency (sustain=10 Hz). The value midi always means that the parameter is to be taken from an appropriate MIDI input.

A.1.4 Synthesis Nodes

ATLAS currently supports four types of synthesis nodes: oscillators, envelope generators, mixers, and amplifiers. Oscillators also have a pm (phase modulation) input which enables it to be frequency modulated by another oscillator or envelope generator. Thus, the current ATLAS version can readily support both additive and frequency modulation synthesis methods. Future versions may also support various digital filters to provide for subtractive synthesis methods as well.

An *Amp* amplifier node simply amplifies incoming samples from a designated signal node. This is accomplished by multiplying each incoming sample by a fixed volume and by an optional additional am (amplitude modulation) factor equal to the current output of another designated node. The amp sample generation rate is therefore equal to that of designated signal node.

A *Mix* mixer node simply *averages* out the input from any number of incoming signal nodes. The mix sample generation rate must therefore be equal to that of the signal nodes with the highest sample generation rate.

As we soon see, osc waveforms, e.g. wave=sine, and env ADSR envelopes, e.g. adsr=linear, are pre-packaged as lists of samples, called *sample tables*. Both osc and env nodes are essentially *sampler* nodes with amp qualities in the sense that they produce amplified sequences of output samples taken from prescribed sample tables. There are distinctions, of course. An osc node, for example, will always use a periodic sample table, referred to here as a wave, whereas an env node will always use a adsr-type sample table containing a segments of samples for each of the three attack, sustain, and release phases. Oscillator sample rates are generally much higher than those generated by envelope nodes, and the oscillator wave is sampled periodically, while the sampling of an adsr sample table is somewhat more sophisticated. Regardless of the differences, their basic function for each sample output is the same. Use an index into the sample table to extract an appropriate sample, amplify that sample by the volume and possibly by an additional am factor, pass the result on to the output sample stream, and finally increment the index by an appropriate value. In an osc node, the index increment size is dictated by the sample generation rate, the oscillation pitch, and the size of the sample table. Similarly, in an env node, the index increment size is determined by the sample generation rate, the attack/release velocity or the sustain frequency, and the number of adsr samples in the current segment. Modulating the index increment value will produce a *vibrato* effect, i.e. a frequency modulation of the output signal.

A.1.5 A Waveform Library

As previously mentioned, all wave and adsr sample tables are pre-packaged and contained in an appropriate waveform library, and the configuration file WaveformLibrary variable (see Figure A.1) identifies the waveform library that is to serve the current ATLAS configuration. Figure A.4 illustrates

```
Wave id=Wave1, Set={Wave1a,Wave1b,Sine};
Wave id=Wave1a, Size=128, Coef={1.0,0.2,0.0,0.3};
Wave id=Wave1b, Size=128, Coef={1.0,0.2};
Wave id=Sine, Size=64, Coef={1.0};
Wave id=Wave2, File=Wave.tbl;
Adsr id=Linear,
  Attack={0.0,0.15,0.3,0.45,0.6,0.8,1.0,0.9},
  Sustain={0.8,0.9,1.0,0.9},
  Release={0.75,0.6,0.5,0.4,0.3,0.2,0.1,0.0}
;
```

Figure A.4: A Typical Waveform Library

what a typical ATLAS waveform library looks like. Each such library may contain any number of periodic wave waveforms, as well as any number of semi-periodic adsr waveforms. Each waveform has a symbolic id, e.g. id=wave1, followed by an appropriate specification describing a sample table containing that waveform.

Periodic waveform sample tables are specified by the number of samples (size=128) followed by any number of Fourier coefficients (coef=$\{1.0, 0.2\}$). Letting n represent the number of samples, h the number of coefficients, a_j the jth coefficient, and s_i the ith sample in the table, then the sample table is generated in two stages. First it is generated by

$$s_i = \sum_{j=0}^{h} a_j \cdot \sin \frac{2\pi i}{n}. \tag{A.1}$$

Then, the waveform samples are normalized so that $\max |s[i]| = 1.0$. Rather than use Fourier coefficients, the waveform library also allows us to specify a file name containing an actual list of the samples themselves. As indicated in Figure A.4, the Wave2 waveform samples are contained in File=Wave.tbl. This enables generating arbitrary waveform files using such tools as *Matlab*, and then incorporating them into the waveform library using the file parameter.

Adsr sample tables, on the other hand, must provide a sequence of samples for each of the three envelope generation phases of Section A.1.3. Samples for the attack phase will be generated only once for each MIDI on event, whereupon the sustain phase is entered automatically, and all sustain samples are sampled periodically. This continues indefinitely until a corresponding MIDI off event, whereupon samples for the release phase will be generated until the end of the sequence is reached. In the adsr example of Figure A.4, the chosen symbolic name is linear due to the linear nature of the waveforms in all three phases.

A.1.6 Aliasing and Wave Sets

Aliasing is a very audible and undesirable off-pitch side-effect produced when the synthesis sample rate does not exceed twice that of the highest spectral content in the output signal. The synthesis sampling rate is, of course, upper-bound by f_s, so if aliasing is to be prevented, then the number of waveform coefficients determines a maximum pitch for any oscillator using that waveform. Letting p denote the required pitch, and h the number of coefficients, then $h \geq f_s/2p$ will cause aliasing. When f_s cannot be increased, then we prefer to use a waveform with fewer coefficients.

To help prevent aliasing, ATLAS waveform libraries support the notion of a wave set, whereby a single waveform id represents a set of waveform alternatives. In the piano.wave1.osc node of Figure A.2, we find that the wave1 wave set was specified as the oscillator waveform. Each waveform in a wave set is expected to have a different number of coefficients, so that for any given pitch, ATLAS can always choose the waveform alternative which has the greatest number of coefficients which will not cause aliasing.

A.2 The MIDI Input

MIDI is an acronym for Musical Instrument Digital Interface, and it has become a well recognized, widely supported standard, for communicating how a musical channel should be or is being played. The MIDI standard includes a physical connector specification, an electrical specification for an online 31.25 Kbaud asynchronous serial interface, a data format for standard MIDI messages, an open-loop, no hand-shaking, protocol for relaying these messages, and a MIDI file format for storing several tracks of prepackaged time-triggered MIDI messages for playback. It was designed to support a wide range of audio devices with arbitrary instrument programs, including song sequencers and drum machines to provide accompaniments, as well as a variety of input controls: e.g. velocity and pressure-sensitive keys, pitch wheels, levers, pedals and switches.

A.2.1 MIDI Streams

A MIDI *stream* is a timed sequence of synthesizer keyboard events, e.g. pressing a given key at a given velocity, altering the key pressure, using the pitch wheel to modulate the pitch, moving a lever, pressing a pedal, and releasing the key. Current digital synthesizers have already demonstrated that velocity or pressure-sensitive keys, pitch wheels, levers, and pedals can indeed be useful for emulating a wide range of both keyboard and non-keyboard instruments.

The current ATLAS implementation supports a single, 16 channel, MIDI stream, but future versions might support several MIDI streams. Appropriate MIDI messages in the stream, dictate the notes to be played by each channel, and also program these channels on-the-fly to sound like any of several pre-determined instruments.

A.2.2 Dynamic Voices

Most instruments can generate more than one note at a time, i.e. playing a three-key chord will generate three concurrent notes. Each note is typically generated by a basic synthesis unit called a *voice*, and the maximum number of concurrent voices in any given synthesizer will always be limited by available resources. In typical hardware synthesizers, the maximum number of voices is even program-sensitive. The Yamaha P-400, for example, has 28 hardware synthesis units for voice synthesis, but some voices, e.g. electronic pianos, electronic organs, accordions, vibraphone, and various ensembles, actually require two and sometimes three such units per voice. Thus, the ability to provide for a typical MIDI stream will highly depend upon how each MIDI channel is programmed.

In the ATLAS context, however the maximum number of concurrent voices is solely determined by the Polyphony configuration variable, independent of the way the MIDI channels are programmed. Note that a sixteen channel MIDI stream may actually require 96 concurrent voices if all channels decide to play three-key chords with both hands at the same time. The allocation of available ATLAS voices to MIDI channels is dynamic in accordance with the immediate needs of each channel. Thus the pool of available voices will typically serve different channels at different times.

	status byte	meaning	data length	parameters	
Voice Messages	8x	note off	2	pitch	velocity
	9x	note on	2	pitch	velocity
	Ax	key pressure	2	pitch	pressure
	Bx	parameter	2	parameter id	setting
	Cx	program	1	program id	
	Dx	chan pressure	1	pressure	
	Ex	pitch wheel	2	14-bit value	
System Messages	F0	system exclusive	variable		
	F1	undefined			
	F2	song position	2	14-bit value	
	F3	song select	1	song id	
	F4	undefined			
	F5	undefined			
	F6	tune request	0		
	F7	EOX (terminator)	0		
Real-Time Messages	F8	timing clock	0		
	F9	undefined	0		
	FA	start	0		
	FB	continue	0		
	FC	stop	0		
	FD	undefined	0		
	FE	active sensing	0		
	FF	system reset	0		

Table A.1: MIDI Messages

A.2.3 MIDI Messages

Each MIDI stream is comprised of a sequence of MIDI messages, each message comprised of a sequence of 8-bit octets (bytes). As indicated by Table A.1, MIDI supports three basic classes of MIDI messages: voice, real-time, and system. All voice and real-time messages are fixed-length.

Voice messages represent channel-specific events, e.g. pressing a key (note on), releasing a key (note off), or switching to another program. Each channel may have any number of concurrent on notes, requiring an equal number of dynamically allocated voices. Each allocated voice is then identified by a channel number and pitch, so that subsequent on or off messages are automatically associated with an already allocated voice if they have the same channel number and pitch. The pitch wheel message is used to modulate the voice pitch, while the parameter message can be used to indicate other lever, pedal and switch-controlled effects such as panning, vibrato, reverberation, sustaining, fading, and more. The parameter message is also used to facilitate a specialized sub-class of voice messages, called *mode messages*, used to control the receiver's overall mode of operation, e.g. turning off all notes, indicating whether local control is enabled, specifying which channels to listen to, and indicating whether voices are monophonic or polyphonic. Mode messages are ignored by ATLAS since ATLAS listens to *all* channels of all MIDI streams prescribed by the current ATLAS configuration file.

Real-time messages serve for synchronizing several MIDI devices in time, sequencer pausing, continuing, watchdog activation and sensing, system resetting, etc. Such messages are particularly necessary when orchestrating several song sequencers and drum machines on several, otherwise uncoordinated, MIDI devices for more elaborate accompaniment. These, too, are ignored by ATLAS since all ATLAS MIDI stream activities are coordinated by other means. In the ATLAS context, real-time messages will be used only to sense that the MIDI interface is alive when no voice messages are being transmitted.

System messages are for controlling device specific capabilities which are not channel specific. Rather than limit the protocol to a common denomination of features, MIDI designers preferred a more flexible approach, allowing manufacturers to incorporate these *system-exclusive* messages and to determine control settings which vary from device to device. Besides providing for variable-length system-exclusive messages with product-specific directives, system messages cooperate with real-time messages in synchronizing sequencers and drum machines, possibly from different manufacturers. Thus, an appropriate system message will inform the song sequencer to begin at a specific *song position* when the next real-time *start* message arrives. All system messages are ignored by ATLAS.

The MIDI protocol also supports a *running status* feature which exploits the fixed message length of all voice messages to avoid resending repetitive message opcodes. All such features must be fully supported. As already mentioned, ATLAS relates only to the more standardized *voice* messages, totally disregarding system-exclusive messages, and those which relate to song sequencers, drum machines, and non-standardized input controls. Nevertheless, the MIDI input processor must recognize the entire repertoire of MIDI messages and features so that it knows what to ignore.

A.2.4 MIDI Events

MIDI voice messages which are supported by ATLAS are henceforth referred to as MIDI events. Figure A.5 shows a typical scenario of MIDI events, as displayed by ATLAS. Each of the top two windows displays MIDI events belonging to a specific channel. Each of the smaller circles denotes a note on event indicating the beginning of a note attack. Attached to each such circle is the note pitch identified by a key and octave, .e.g. the C3 notes at time 0 in channel 0 identifies key C in the third octave. Each key is essentially a half-tone in a 12 half-tone per octave scale, in which each half-tone is identified by one of the symbols A, A#, B, C, C#, D, D#, E, F, F#, G, and G#. In the above scenario, channel 0 has two simultaneous notes, C3 and D3, starting at time 0. To the right of each circle we find a tail indicating the duration of the note, with the end of the tail indicating the arrival of a note off event. To the right of each tail is the integer identifying the voice allocated to synthesize the tail's note. Near the upper left corner of each channel window are larger circles without tails that identify the program events that determine the set of instrument synthesis methods to be chosen from in each channel when generating a note.

A.2.5 MIDI Files

The notion of time and the timing relationships between several parallel MIDI streams are sufficiently implicit when using online interfaces. When using pre-recorded MIDI files, however, these issues must be encoded within the MIDI file format. To accomplish this, each MIDI message in this for-

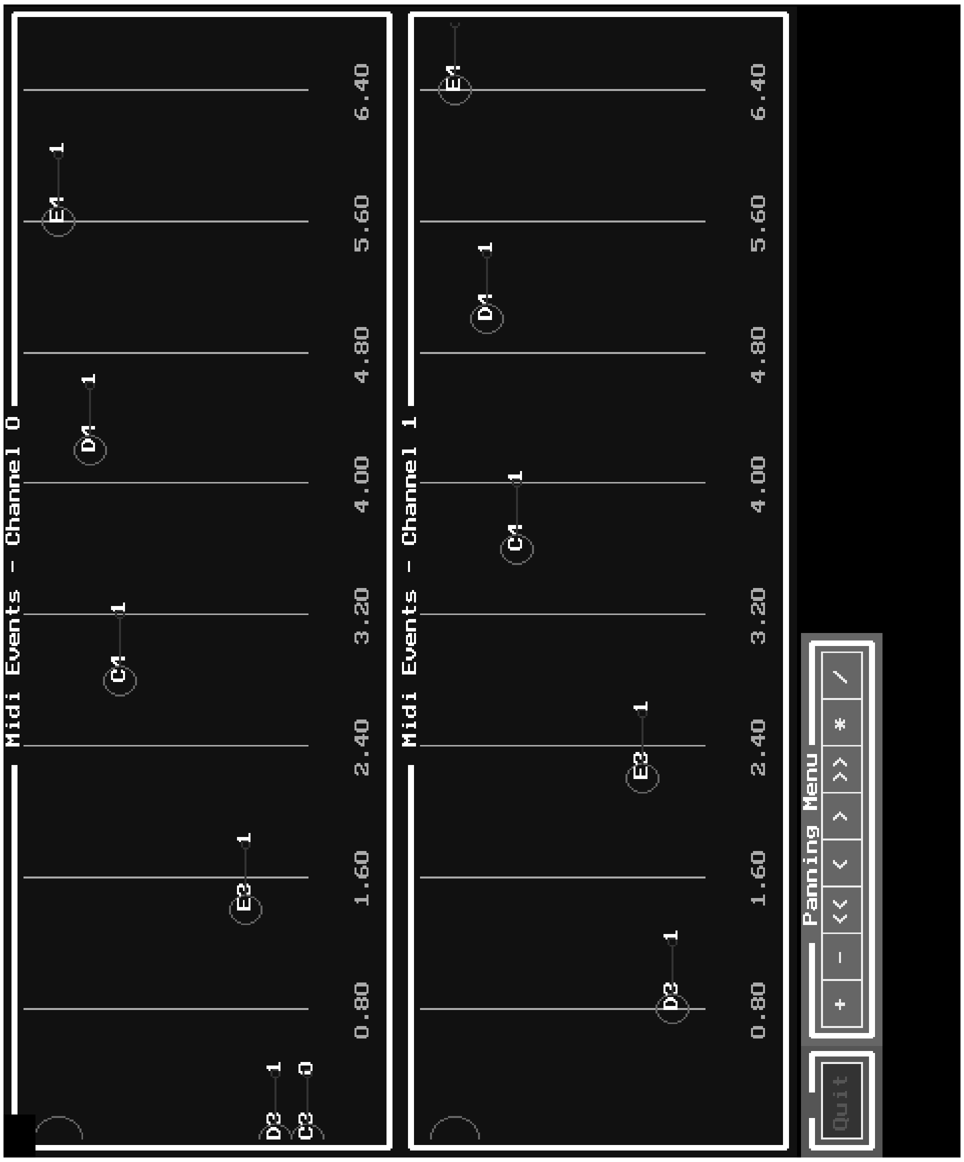

Figure A.5: MIDI Events

mat is prefixed by a *differential* time stamp, called *delta-time*, indicating when the following message should be transmitted, relative to the previous message. Offline MIDI files may also contain non-MIDI messages called *meta-events*, containing additional music-playing directives which are not MIDI supported. MIDI file delta-times may use any of several standard time-referencing schemes. To be economical, they are encoded using a standard called *variable length quantity.*

A.2.6 MIDI Time Frames

To realistically emulate an online MIDI interface, MIDI octets are analyzed and handled only when preceding time-stamps indicate that these octets would have been available if an online MIDI interface were being used. In both cases, octets are treated one by one using an appropriate state-machine, just as it would have to be when using an online MIDI interface. In both cases, available octets are treated at predetermined TimeFrame intervals (see Section A.1), just as it would have to be when using a MIDI file. Thus the only difference between the two cases is the way in which available octets are accumulated for each time frame. When using an online interface, octet accumulation is interrupt-driven. When using a MIDI file, available octets are read from the file during the previous time frame, taking effect at the onset of the following frame.

A.3 Real-Time Synthesis

As illustrated by Figure 1.2, a typical audio outlet consists of a clock-driven FIFO queue which automatically feeds a speaker at a software controlled sampling rate, f_s, via a S&H (sample and hold) circuit and a D/A (digital-to-analog) converter. Playing a concurrent set of digital voices requires the parallel generation of steady sample streams for each voice in a manner which enables them to be mixed and fed via such FIFO queues to one or more D/A audio outlets at appropriate sample rates. In a typical electronic synthesizer, voice sample rates are fixed, voice synthesis methods are determined by the channel programs, and a sudden demand for too many voices will result in rejection on a FCFS basis.

In ATLAS, all voice synthesis is carried out in software, so that voice combinations are limited only by the computational resources provided by the current, possibly distributed, runtime platform. As in conventional synthesizers, voices are dynamically allocated to MIDI channels, clearly determining the program to be emulated by that voice. Unlike conventional synthesizers, however, an ATLAS voice may use any of the alternate synthesis methods provided for that program by the currently configured **synthesis method** library, and it can also adopt a lower voice sample rate to reduce the computational load. When overloaded, ATLAS is required to exploit these inherent flexibilities to choose those voices, synthesis methods, and sample rates which will maximize the overall SNR for each platform, rather than reject work loads on a FCFS basis. Thus louder volume notes will almost always be preferred over lower volume notes, and a note in an attack phase will receive preference over a note in a release phase.

A.3.1 Sample Rate Optimization

As already mentioned, the computational load of any given synthesis method can be reduced by minimizing the sample generation rates for each of the nodes within the synthesis method. This minimization of node sample rates will henceforth be referred as *sample rate optimization*, or SRO. By default, all synthesis nodes for a given method share a common sample generation rate which is equal to f_s. When using SRO, the method resource requirements can be considerably reduced, depending upon the current pitch, the number of waveform coefficients, and a few other factors. Generally speaking, the sample generation rate, f_v, for any given voice, v, is lower bound by the *sampling theorem* which requires that the sample rate always exceed twice that of the highest spectral component in the voice output, so a higher pitch will always require a higher sampling rate. The rules for sample rate minimization must also prevent aliasing and ensure that the output sample rate for each mode will always exceed that of any of the inputs to that node, so that no resolution is lost. To minimize voice mixing overheads, we also require that all f_v and node sample rates be integral divisors of f_s. Thus each synthesis method in the synthesis library essentially represents two possible modes, with and without SRO, and each method requires a distinct SNR reward for each mode. Thus the Add3 method of Figure A.2 has two SNR reward values, 0.4 with SRO and 0.9 without.

Switching to another synthesis method while generating a note will almost alway cause a sudden phase shift which is audible and disturbing. Consequently, once a given synthesis method has been chosen, each voice remains committed to its current method until it completes its current note[2]. Tran-

[2]Future versions may find ways to gradually phase in a new method while the older method is phased out, but this can only be done when there is adequate capacity to

sitions between SRO and non-SRO modes within a given synthesis method *cannot* produce a sudden phase shift, so they are allowed at any time, even in the midst of playing a given note.

A.4 The Graphic User Interface

ATLAS must provide a menu-driven interface which lets the operator monitor and control ATLAS activities via a mouse or keyboard. When carrying out real-time synthesis, current operator options are limited to pausing, accelerating, and slowing down. Future ATLAS versions will also enable the operator to dynamically alter available resource capacities to test for fault-tolerance, etc. The menu-driven interface requires that the system respond adequately to the operator mouse and keyboard even when carrying out real-time synthesis. A mild response might require only a relocation of the current mouse cursor. A more significant response might entail repainting the menu items to clearly mark the menu item which will be selected if an appropriate mouse button is clicked or an appropriate key is entered. Significant and varying levels of operator-instigated sporadic and aperiodic activities are thus introduced into the real-time requirements.

A.4.1 Operator Events

Figures A.3 and A.5 illustrate what is meant by an "operator" event. In Figure A.3, we see that the Wave1a item is highlighted (darker) in a menu which lets the operator view various portions of the current program and waveform library. Synthesis methods selected for viewing are displayed in the upper left window. Selected programs, nodes, and wave sets are viewed in the center left window. Selected waveforms are viewed in the bottom left window. The menu on the right lists all the currently defined programs, methods, nodes, wave sets, and waveforms. Mouse movement, mouse clicks, and pressed keys all produce operator *events* which must be processed by the system. To highlight a given menu item, the operator may click a mouse button or use any of the arrow keys, tab keys, or illuminated hot keys. The selected item is displayed in an appropriate window only if the mouse is double-clicked or if the carriage return key is entered. Typical operator events in this example require updating the mouse cursor, redrawing highlighted and non-highlighted items when the selected item has been altered,

concurrently support *both* methods during the interim period.

and upgrading appropriate display windows when selected items have been activated.

The same operator events serve to select and activate the items in Figure A.5. Here we have two menus to choose from at the bottom of the screen. The left menu has a single item for returning to the main menu. The right menu is a panning menu which lets the operator zoom and pan when displaying the MIDI events in a given MIDI file. The + and - items are used to zoom in and out, the < and > items are used to pan left and right, the * and / items are used to double or halve the panning step size, and the << and >> items are used to jump to the file beginning or end. When activating a panning menu item, the system response will usually require reprocessing the entire MIDI file while displaying only those MIDI events which belong to the selected time segment. Thus aperiodic responses to operator events may be quite intensive.

A.4.2 Offline Displays

Most GUI displays serve offline activities, such as configuring the application, testing and debugging facilities, and profiling platform and application activities. Figures 8.5, 8.6, 8.7, 8.8, and 8.9, for example, provide the set of offline displays that serve the platform-dependent profiling sequence discussed in Section 8.2.5. Nevertheless, current GUI requirements also include real-time responses ATLAS to a small set of operator events, as well as the real-time display of MIDI events and the audio output volume. These real-time responses are the subject of the following subsections.

A.4.3 Real-Time Options

To make it interesting, ATLAS real-time synthesis also supports two toggle options, yes/no graphics and yes/no SRO, which can be manipulated at any time during the real-time synthesis. The no graphics option enables us to sense to what extent the system gain performance is affected by the need to support a graphic display while carrying out real-time synthesis. The no SRO option enables us to sense the affect of reducing the system's options to adapt when in the midst of playing a note. The commitment to maintain a given synthesis mode throughout a note duration should encourage us to develop more sophisticated policy-making schemes which must anticipate all possible state transitions while it is still committed.

A.4.4 The Real-Time Display

Figure 8.1 shows a typical ATLAS display during real-time synthesis. The top two windows serve to display each MIDI event as it is being processed. Below them we find the Mixed Output window which displays the minimum and maximum sample values output to the audio outlet during each MIDI time frame. These min/max values are more extreme in the beginning because of the two concurrent notes in Channel 0.

Beneath these windows we find a menu with toggle-like items to enable or disable the no graphics option, to start or stop the real-time synthesis, and to enable or disable the SRO option (opt on). ATLAS requirements include that operator events may select any of the menu items and may activate any of the above real-time option items without interrupting real-time synthesis.

A.4.5 Platform Emulation

To enable more extensive experimentation on any given platform, ATLAS should also support a *platform emulation* mode where time is scaled up or down by a given factor, TimeScale, to simulate what would happen when running on a platform from the same family with an arbitrary number of shared-memory processors and arbitrary capacities. Generated samples are recorded in a standard RIFF format used for recording audio samples for playback. This enables ATLAS to later replay these files to hear how it would have sounded. This platform emulation mode is real-time in the sense that it must emulate the platform in the time dictated by the TimeScale, e.g. a 20 second recording must be produced in 20· TimeScale seconds. Thus ATLAS must also take into account differences in the time needed to feed the FIFO queue and the time needed to record an RIFF file. Default values for TimeScale, the number of emulated processors, and their subsequent capacities, may also be prescribed by an ATLAS configuration file. An appropriate menu will provide interactive control over these values.

As illustrated by Figure 8.2, ATLAS must also consider differences in the emulation display. Here we see that *all* output samples are displayed, instead of just extreme min/max values per time frame. The note amplitude envelopes are much clearer in this display. We also see that the control menu has two new items, Debug and Replay. The Debug item enables us to process one time frame at a time and display the output buffers of all active synthesis nodes at the end of each time frame. The Replay item lets us replay the RIFF recording produced by the platform emulation. During this Replay mode, we can pan and zoom into arbitrary segments of the RIFF recording, listen to them, or record them in a Matlab-compatible format for further Matlab analysis.

Note that the platform emulation mode also allows us to run ATLAS on platforms which lack audio outlets. RIFF files are then readily transported to other platforms for replay via standard tools, or we can analyze the output visually or by other means, such as with Matlab.

A.5 Online Trace Recording

ATLAS requirements include the online, time-stamped, recording of trace information to ascertain functional and temporal correctness, and to enable the offline analysis of system gain performance. All events and system responses must therefore be recorded in a minimal intrusive manner. The tracing mechanism must also enable us to follow interrupt-driven activities, something which is rarely supported by conventional debuggers.

Figure 8.10 depicts a typical trace segment after it has been translated into a textual format and analyzed by the ATLAS performance analyzer. As we can see, all binary trace records are 32-bit time-stamped with a sub-microsecond resolution so that accurate execution times can be computed. Each trace record has a 16-bit *id*, identifying the program location which produced it, along with an additional 16-bit *value* which can be used in arbitrary ways. Thus each trace record is 8 bytes long. As illustrated by the example, each trace id belongs to a specific class which is identified by the upper-cased name following each time-stamp. Some classes serve to measure job execution times, some verify that hard deadlines are met, and some serve to record operator events and all automated decisions. Each achieved SNR value is compared, offline, with that which would have been obtained by QDP with a very fine bandwidth quantum parameter (see Section 6.2). According to Section 7.3, this should provide us with a good ϵ-approximate evaluation of actual ATLAS gain performance.

A.6 Portability

ATLAS is initially targeted to run on two families of platforms but should readily be ported to addition platform families, as well. The top-layer models of this thesis are currently limited to shared-memory multiprocessor platforms, but we wish to extend our research to distributed platforms. Thus

ATLAS might eventually be designed to run on a wider variety of platform architectures: single processor, shared memory, and distributed, with arbitrary numbers of processors and arbitrary combinations of processing speeds.

To demonstrate the feasibility of distributed musical synthesis, a preliminary, non-adaptive, ATLAS implementation ran on a pair of distributed UNIX-based Silicon Graphics Indigo machines, with one machine generating the audio samples and the other machine feeding the audio outlet. Our current version is designed to manage without preemptive scheduling so that it can run also on conventional DOS-based PCs, with arbitrary x86 processors and processor speeds. Future versions should also support preemptive scheduling and multiprocessor machines as well.

Appendix B

ATLAS Implementation

Our initial VaPoR version has been fully implemented for both PC and Indigo platforms, but only the PC version has been debugged and tested[1]. As described in Section 8.2.7, all differences between the PC and Indigo versions have been isolated within the VaPoR machine library. Cross-platform GUI libraries such as OpenGL, and virtual machines like Java were not yet available when ATLAS development began, so this aspect of the top-layer architecture had to be developed from scratch. A detailed description of VaPoR design is beyond the scope of this document, so we focus only on the more innovative aspects of our implementation.

A major challenge, here, was to devise a common interface for both VaPoR machines which could be readily translated into available services on each platform. Of the 49.3K lines of ATLAS source code, 7.7K lines belong to the dual-platform VaPoR machine source files. Internal compiler directives ensure that a platform-specific VaPoR implementation is automatically produced for each of the supported platforms. Wherever possible, the VaPoR machine implementation will simply translate VaPoR services into appropriate operating system calls. In other cases, VaPoR provides these services on its own. The VaPoR GUI, for example, uses standard Xlib routines when running on an X11-based platform, while providing its own mouse driver and windows manager when running on an MS-DOS platform. For standardized keyboard codes and mouse events, Xlib conventions are translated into those of MSDOS. The VaPoR sound interface uses the standard Indigo audio driver when running on an Indigo platform, while pro-

[1]Several portions of the Indigo VaPoR machine have already been tested in a preliminary ATLAS version.

viding a customized 8-bit DMA-driven Sound Blaster driver for PC-based platforms.

A detailed description of the current ATLAS implementation is beyond the scope of this thesis. We therefore focus, here, on only a few of the more innovative aspects of our system. Each platform, for example, has a different time resolution, yet the notion of time is essential to real-time applications, so it must somehow be supported by VaPoR in a platform-independent manner. This prompted us to develop the notion of *virtual time*, which we discuss in Section B.1.

Another major achievement was the development of a substantial, yet simple, GUI paradigm which was readily implemented for both platforms. To maximize the variety of supported platforms, we chose to base our Indigo VaPoR machine on the X11 graphics library. X11 overheads can be very significant, so our challenge, here, was to select a specific mode of operation which would provide our basic needs while minimizing these overheads. Another problem was that the standard X11 was not designed to serve real-time applications, so we also had to find a way to control the scheduling of the event-driven X11 activities. For our PC VaPoR machine, on the other hand, we initially considered adopting a Windows-based environment, but it was eventually ruled out due to its lack of provisions for real-time scheduling. We therefore had to manage with a DOS-based graphics library provided by Borland C++, which had only minimal support for windows and no mouse support whatsoever. Our GUI paradigm is discussed in Section B.2.

The Indigo provides an appropriate FIFO-based interface for audio output, but available Sound Blaster drivers did not. So we had to develop an appropriate Sound Blaster driver for our PC VaPoR machine, which would emulate a FIFO-based audio outlet while exploiting the DMA capabilities of the Sound Blaster card. More on this is described in Section B.3.

Both VaPoR and PReTSL were developed in a manner which cover all of ATLAS's needs, but which could readily be reused for other applications. Thus, the VaPoR libraries can readily serve as a common machine-independent platform for a wide range of musical and non-musical applications. A brief survey of current VaPoR capabilities is provided by Section B.4. All VaPoR and PReTSL libraries were implemented in C++ to exploit its object-oriented capabilities. It has been widely debated whether object-oriented programming is appropriate for real-time applications. Initial thoughts on this matter appear in Section B.5.

B.1 A Virtual Timer

Timing mechanisms and resolutions differ from one platform to another, yet VaPoR requires a *uniform* mechanism for generating high resolution time stamps and for measuring time. The strategy chosen allows for different resolutions while ensuring that the time scale remains uniform. A VaPoR time stamp is a 32-bit entity providing a relative notion of elapsed time. VaPoR time stamps are *virtual* in the sense that actual time stamp values may significantly differ from one platform to another, but time intervals measured between two time stamps from the same platform will always be as identical as possible. Thus a PC time stamp has a time resolution of 0.8384 microseconds because its 8254 timer uses a 1.193 MHz clock, while UNIX time stamps will have a resolution of 1 microsecond because this is the resolution provided by the standard UNIX timeofday() service. Actual time intervals between time stamps are obtained by appropriate VaPoR routines which take the varying platform resolutions into consideration.

Time stamp accuracies will almost always be lower than the time stamp resolutions. On conventional UNIX platforms, time accuracy will usually be 5 to 10 milliseconds. Standard MS-DOS time accuracy is only 55 milliseconds, so the PC VaPoR machine accesses the 8254 timer directly to obtain accuracies of 10 microseconds and more. Each 32-bit time-stamp comprises two 16-bit fields, The lower field containing the two's complement of the current 8254 system timer counter status, and the higher field containing the current value of the internal interrupt-driven *tick* counter which is incremented with each system timer end-count. The PC VaPoR machine implementation was carefully designed for minimal service overhead and for handling race conditions which can occur when counter end-count interrupts occur while the timer counter is being read.

As illustrated in Figure B.1, the VMCLOCK TIMER class is responsible for all virtual time services. The stamp method creates time stamps with the highest available resolution, while the methods below it serve to measure the number of elapsed seconds between any two time stamps. A private LastTime timestamp is updated with each stamp call so that elapsedSecs can adopt it as the reference time stamp if no other is provided. When running on a 486DX/33 MHz PC, the TIMER cost of creating a time stamp was found to be less than 10 microseconds. Thus TIMER provides time measuring services which are a thousand-fold more precise than those provided by standard DOS services. Only one TIMER object should be created, so that a global TIMER is provided.

```
typedef long TIMESTAMP;          // must be signed
class TIMER {
  TIMESTAMP LastTime;
public:
  TIMER();                       // TIMER constructor
  ~TIMER(void);                  // TIMER destructor
  TIMESTAMP stamp(void);
  float stampSecs(TIMESTAMP t); // since TIMER construction
  float elapsedSecs(TIMESTAMP since, TIMESTAMP til);
  float elapsedSecs(TIMESTAMP since); // from since to now
  float runtimeSecs(void); // since TIMER construction
};
extern TIMER timer; // global TIMER object
```

Figure B.1: The TIMER Object Interface

B.2 The GUI Paradigm

To minimize graphics overhead, the VaPoR machine GUI support is limited to a single centralized CONSOLEQ object for collecting mouse and keyboard events and to any number of *tiled* WINDOW objects for graphic and textual display. GUI classes in the portable VaPoR library also support multiple menus, pannable graphs, type-sensitive editors, and more.

B.2.1 GUI Input

As illustrated by Figure B.2, the CONSOLEQ emulates a single input queue for mouse and keyboard events in a manner which lets the application process mouse and keyboard events in an independent or integrated manner, in accordance with application needs. Figure B.3 illustrates how integrated mouse and keyboard events might be typically processed. The ready method provides us with the current queue status, using specific bits to identify the nature of the first queue event. The getMouse and getKey methods can then be used to process these events. The what field identifies the nature of a mouse event: mouse button up or down, mouse movement, or auto mouse events. All mouse events are also internally time-tagged, so that double-clicks can be identified. Both of these methods will fetch the appropriate event type even if the event at the head of the queue is of another type. Thus the application may ignore mouse or key events in accordance with its priorities and needs. The keyPressed method is more useful than ready when

```
class CONSOLEQ
{
public:
  CONSOLEQ();
  ~CONSOLEQ(){done();};
  void init(int qsize);    // setting the input queue size
  WORD ready(void);        // returning the queue state
  BOOL keyPressed(void);   // any key events in queue?
  BOOL getMouse(MOUSEEVENT &); // get first mouse event
  WORD getKey(KEYEVENT &); // returning 0 if none
  WORD getKey(void);       // get first key event
  void restoreKey(void);   // restore last read key back to queue
  void beep(void);         // generate beep
  void suspend(void);      // suspend mouse events
  void resume(void);       // resume mouse events
  void doubleDelay(int ticks); // for double click
  void mouseReverse(BOOL on);  // reverse mouse buttons
  void done();             // deallocate input queue
};
extern CONSOLEQ console;
```

Figure B.2: The CONSOLEQ Object

interested only in keyboard events. Events will be lost only if the input queue size set by init is too small. The input queue size can be adjusted at any time by subsequent init calls.

CONSOLEQ behavior is platform-independent even though implementation differs from one platform to another. All key codes and mouse events, for example, conform to common MS-DOS conventions. Internal X11 key codes are therefore converted into those which are provided by the standard BIOS calls. Mouse coordinates are provided in pixels, relative to a full display screen with an origin of (0,0) at the top left corner, so all key and mouse events are propagated to an X11 root display window which is made to appear just like a 640x480 VGA screen. Even X11 mouse event times are translated from milliseconds into PC ticks so that double click detection can be carried out in the same manner.

CONSOLEQ implementation was somewhat more demanding on the PC VaPoR machine, since the Borland C++ graphics library provided no mouse support. Regardless of when CONSOLEQ is called, a PC mouse device transmits its internal state via a serial communications port at a given baud rate. The standard microsoft mouse driver at software interrupt 0x33 re-

```
if ((WORD mask=console.ready())>0) {
  if ((mask & KEYMASK)>0) {
    WORD keycode = console.getKey(key);
    switch (keycode) {
      case HOMEKEY  : ... break;
      case CR       : ... break;
      case LF       : ... break;
      case RIGHTKEY : ... break;
      case PGUPKEY  : ... break;
      case SHIFTTAB : ... break;
      ...
    } // switch(keycode)
  } // key event
  else {
    MOUSEEVENT mouse;
    console.getMouse(mouse);
    switch (mouse.what) {
      case MOUSEUP   : ... break;
      case MOUSEMOVE : ... break;
      case MOUSEDOWN : ...
        if (mouse.where ....) ...;
        if (mouse.doubleClick) ...;
        ... break;
    } // switch (mouse.what)
  } // mouse event
} // if (console.ready)
```

Figure B.3: Typical CONSOLEQ Consumption ceives these mouse transmissions to note changes, update cursor display, and possibly activate the CONSOLEQ user handler when predetermined conditions arise. In the Indigo VaPoR machine, the CONSOLEQ interface is translated into appropriate qdevice and qread routines and the following X11 inputs are queued: the RIGHTMOUSE, MIDDLEMOUSE, LEFTMOUSE buttons, the MOUSEX, MOUSEY valuators, the TIMER0 timer device to gain a sense of time, and the KEYBD. Background activities are accounted for on both platforms by extreme mouse activity during platform noise profiling. The processing of input events is, of course, scheduled by the application. As we can well see, a global CONSOLEQ object named console is automatically *constructed* and the CONSOLEQ user handlers installed, so that mouse input is automatically made available to the application. It is also automatically *destructed* when the program exits in an orderly fashion via exit().

B.2.2 GUI Windows

The WINDOW class provides any number of fixed and tiled framed windows sharing a single parent application window, referred to as the *display* window. Windows are *fixed* in the sense that window moving and resizing cannot be carried out by an operator unless this capability has been programmed within the application. Windows are *tiled* in the sense that no support is provided for redrawing that part of a window which has been covered by an overlapping window. This fixed and tiled windows paradigm is simple enough to implement for any given platform, especially after we have already done so for our PC VaPoR machine with only minimal support from the Borland C graphics library. It also insures that window maintenance overhead is minimized and controlled since we need not cater to window refresh activities which might otherwise be required by operator events such as moving, resizing, or selecting a specific window.

Figure B.4 depicts the more essential portions of the VaPoR WINDOW interface. As we can see, the parent display window is opened, cleared, and closed by respective openDisplay, clearDisplay and closeDisplay functions, and only a single text font is supported, so that all text has a global textWidth and textHeight. All windows also have a common fixed border width, provided by borderWidth. Child windows are dynamically allocated by the init method and deallocated by the done method. As indicated by the init method, each window has a centered caption at its top border, and it fills a fixed frame within the parent window with given default foreground and background

```
typedef float DEGREES;
void openDisplay(void);  // opening the GUI display
int  borderWidth(void);  // border width in pixels
int  textHeight(char *); // max font height in pixels
int  textWidth(char *);  // max font width in pixels
void clearDisplay(void); // clearing the GUI display
void closeDisplay(void); // closing the GUI display

class WINDOW {
public:
  void  init(char *caption, BOX frame, int bkgnd, int foregnd);
  WORD  maxY(void);   // max relative Y coordinate within window
  WORD  maxX(void);   // max relative X coordinate within window
  void  open(void);   // open window
  void  close(void);  // close window
  void  clearScreen(void);  // clear screen
  void  setColor(int);      // set current foreground color
  int   color(void);        // get current foreground color
  void  setBackground(int); // set current background color
  int   background(void);   // get current background color
  BOOL  inBox(BOX,POINT);   // is absolute POINT in relative window BOX
  void  clearBox(BOX);      // clear BOX within window
  void  drawBox(BOX);                          // draw BOX
  void  drawLine(POINT, POINT);                // draw line
  void  drawCircle(POINT,int);                 // draw circle
  void  connectCircles(CIRCLE,CIRCLE);         // connect circles
  void  drawArc(POINT,DEGREES,DEGREES,int,int); // draw arc
  void  drawText(POINT *at, char *text);       // draw text
  void  drawCenterText(POINT at, char *);      // draw centered text
  void  newLine(POINT *at);                    // emulate CR+LF
  POINT fPoint(float x, float y);  // x=[0,1]; y=[-1,1];
  POINT uPoint(float x, float y);  // x=[0,1]; y=[0,1];
  void  done(void);
};
```

Figure B.4: VaPoR Windows Support

colors. Each window has a (0,0) origin at the top left corner. The maxX and maxY methods provide the maximum pixels coordinates within the window, which will always be equal to the window frame size minus its border widths.

Windows can then be opened and closed an arbitrary number of times. Once opened, the WINDOW class provides a significant variety of methods for drawing text, boxes, lines, circles, and arcs within the window. Each window supports a notion of a current drawing point (cursor) as well as current foreground and background colors which can be altered at any time. These capabilities are exploited by higher-level classes which support colorful multiple menu blocks and pannable and zoomable graphs, such as those illustrated in Figures A.3, A.5, 8.1, 8.2, 8.9, and the other ATLAS displays in this chapter. The WINDOW class also provides an inBox method for testing whether the mouse cursor currently falls within a given box, as well as fPoint and uPoint methods for converting arbitrarily scaled x, y coordinates into appropriate pixel coordinates within the window. Another nice WINDOW feature is that it enables the application to maintain any number of concurrent *virtual* cursors for drawing in different window areas. Most of the drawing methods are also highly overloaded in the sense that each has several variations. Thus drawText has several methods for drawing text at the default window cursor or at an application-provided virtual cursor, using a specific color or the default window color. Only a few of these variations appear in Figure B.4 for clarity.

In the PC VaPoR machine, the display window covers the entire VGA screen. In the Indigo VaPoR machine, the parent display is automatically created to be equivalent to a typical VGA screen screen size. Borrowing from the X11 terminology, all keyboard and mouse events are *propagated* to the parent window and handled by a central event handler. In the DOS environment, this propagation is automatically carried out by the CONSOLEQ object. In an X11 environment, CONSOLEQ and WINDOW classes collaborate in the sense that CONSOLEQ implements a DOS-compatible event queue associated with the WINDOW root window, while WINDOW ensures that all events are indeed propagated to it.

B.3 Platform-Independent Audio Support

To achieve platform-independence, the VaPoR audio outlet has been modeled as a two-staged FIFO as depicted in Figure B.5. Each audio device may provide a different sample resolution and may use a different sample

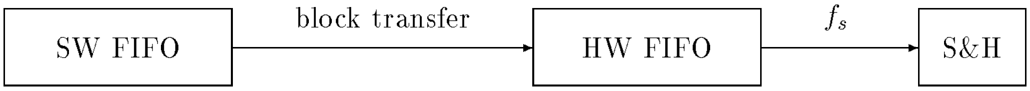

Figure B.5: Two Staged Audio Model

```
class SOUND {
public:
  SOUND();
  ~SOUND(void);
  void init(int buffersize);      // allocating SW FIFO
  void record(char *path, HZ hz); // for RIFF recording
  void on(int dmasize);           // for audio output
  HZ   setSpeed(HZ requestedhz);  // setting output sample rate
  long sampleRate(void);
  void put(FSAMPLE sample);       // outputting floating point sample
  int  filled(void);              // HW FIFO fill
  int  fillable(void);            // HW FIFO avail
  void pause(void);
  void resume(void);
  void flush(void);               // flushing SW FIFO to HW FIFO
  void off(void);                 // ending RIFF/audio output
  void done(void);                // deallocating SW FIFO
};

extern SOUND audio;
```

Figure B.6: VaPoR Audio Support

encoding standard. Rather than conform to a specific standard, the application generates floating point samples and it uses the VaPoR interface to feed these samples into the first stage SW FIFO. VaPoR encodes these samples in accordance with the needs of the current platform and then transfers them into a final stage HW FIFO. This HW FIFO can either be an actual audio device or an RIFF file, or both. RIFF encoding may actually differ from the encoding scheme required by the audio device on the same platform. Thus an ATLAS application can carry out musical synthesis without having to be concerned about the nature of the current platform and without distinguishing between real-time and platform emulation modes.

Figure B.6 depicts the VaPoR machine audio interface. As previously mentioned, the same methods can be used for audio output or RIFF recording. The SW FIFO capacity is set by init and the HW FIFO sample rate is set by setSpeed. The actual audio device might not be able to comply with the requested speed, so it will choose the highest available speed which does not exceed the requested speed, and this speed is returned on the spot by setSpeed and upon request by sampleRate. When using the put method to output samples, one must take into account that put will return only after it has successfully entered the sample into the SW FIFO. If the SW FIFO is full, VaPoR will automatically activate flush to transfer the SW FIFO contents to the HW FIFO. The flush method will always use an block-transfer mechanism, e.g. memcpy, so it can only do this if the HW FIFO has enough room, e.g. if fillable is not less than the SW FIFO capacity. The application should therefore make explicit use of the fillable and flush methods to ensure that these transfers are carried out in a timely manner.

The two-phased model was readily implemented for both PC and Indigo machines despite significant differences. The 16-bit Indigo samples are signed while 8-bit Sound Blaster samples are unsigned. When generating RIFF recordings, we adopted the 16-bit format to maximize the audio quality. Whenever possible, VaPoR actually does without a SW FIFO altogether. Thus the put method actually transfers samples into the Indigo HW FIFO directly, and the SW FIFO is only used for RIFF recording. In the PC machine, we had to develop a customized Sound Blaster driver which flushes the SW FIFO into a cyclic DMA buffer, from which the Sound Blaster card is fed by an automatically repeated DMA mechanism. The cyclic DMA buffer size is dictated by the on method and it must be at least twice that of the init buffer size, yet no greater than 32 KB. For greater deadline tolerance, this DMA buffer size may actually be several times the size of the SW FIFO capacity, but the greater it is, the larger the T_f, i.e. the poorer the response time (see Section 8.2.3). DMA and Sound Blaster interrupts are used to facilitate the DMA driven mechanism and to keep track of the output process so that filled and fillable are always known. It is these filled values which appear in the SAMPLES trace events of Figure 8.10. It is up to the application to ensure that SW FIFO samples are provided at an adequate rate such that the HW FIFO never underflows. VaPoR can at least detect underflows and keep an account of them.

```
channel=0
time=  0.000 (dtime=0.000) htone=A3, attack=70, on
time=  0.250 (dtime=0.250) htone=E5, attack=80, on
time=  0.500 (dtime=0.250) htone=A3, attack=0, on
time=  0.500 (dtime=0.000) htone=E5, attack=0, on
time=  0.500 (dtime=0.000) htone=A4, attack=70, on
time=  0.500 (dtime=0.000) htone=A5, attack=80, on
time=  0.750 (dtime=0.250) htone=A5, attack=0, on
time=  0.750 (dtime=0.000) htone=C6, attack=80, on
time=  1.000 (dtime=0.250) htone=C6, attack=0, on
time=  1.000 (dtime=0.000) htone=B5, attack=80, on
time=  1.250 (dtime=0.250) htone=B5, attack=0, on
time=  1.250 (dtime=0.000) htone=E5, attack=80, on
time=  1.500 (dtime=0.250) htone=A4, attack=0, on
time=  1.500 (dtime=0.000) htone=E5, attack=0, on
time=  1.500 (dtime=0.000) htone=G#4, attack=70, on
time=  1.500 (dtime=0.000) htone=B5, attack=80, on
```

Figure B.7: VaPoR Textual MIDI

B.4 Current VaPoR Services

As already demonstrated by the VaPoR machine classes previously described, the primary VaPoR objective is to provide predictable platform-independent real-time services so that real-time applications can achieve source-level portability. This requires an abstraction of the platform services generally required by real-time applications. As ATLAS developed, VaPoR was extended to cater also to the special platform requirements of many musical applications, with the intent that it should thus serve the Technion Music Lab as a growing repository for useful portable objects. Such a VaPoR library would thus provide the *glue* needed to ensure that Music Lab projects could cooperate.

Following is a brief overview of the primary VaPoR library modules and the classes they contain. VLTRACE provides the minimal-intrusive real-time trace mechanism described in Section 8.2.6. VLMENU provides high-level GUI classes which support multiple-window menus of typed items such as those depicted in Figure 8.5 including the panning menu of Figure A.5.

The VLMIDIB and VLMIDIT modules provide basic MidiIn and MidiOut classes for MIDI input and output. They provide both binary and textual implementations for each of these classes. VLMIDI can then provide a very useful utility program for viewing, translating, merging, splitting, and filtering both binary and textual MIDI files. There is no accepted standard for textual MIDI files, so we devised something simple which we found to be useful. Figure B.7 shows a segment from a typical textual MIDI file which was automatically translated by VLMIDI from a binary MIDI file which plays Bach. These textual files are readily edited and then converted back to the standard binary format via VLMIDI for ATLAS synthesis.

VLRIFF provides classes for reading, writing, viewing and panning RIFF files. A nice VLRIFF feature is that arbitrary RIFF segments are readily output via VMSOUND so that we can hear specific portions of the RIFF file being viewed. Arbitrary RIFF segments can also be output to standard Matlab files so that they can be imported into a Matlab environment for spectral analysis.

As indicated by the configuration file of Figure A.1, the program library of Figure A.3, the waveform library of Figure A.4, the trace file of Figure 8.10, and the textual MIDI file of Figure B.7, ATLAS must process a significant number of symbolic textual files. VLPARSER and VLSYMTBL provide very handy parsing and symbol table classes for parsing textual files and for managing dynamic symbol tables. The dynamic symbol tables are hash-indexed for fast symbol table queries. They are also designed in a manner which enables the memory consumption of symbol tables to be predicted precisely. The symbol tables employ a two pass scheme in order to minimize memory requirements. These parsing and symbol table classes are used extensively by the VLCONFIG, VLWAVES, and VLPROG modules, to input the ATLAS configuration file and to input, access, and view the waveform and program libraries.

VLNODES provides the basic synthesis node class as well as the inherited oscillator, amplifier, envelope generator, and mixer classes which comprise the various synthesis methods. VLUNITS provides a special class for converting among pitch units, whether in hertz, halftones, waveform phase increments, or others.

VLHEAP provides a very useful class for maintaining an indexed event heap. These event heaps are essential to the G^2 meta-control algorithm as well as to the processing of multiple-track MIDI files. These heaps are indexed to ensure that the removal of an arbitrary event in the heap can also be accomplished in $O(\log N)$ complexity where N denotes the number of events in the heap. VLTOKEN provides a class for dynamic voice management, and VLALLOC supports a dynamic memory management scheme which is more deterministic than standard `malloc` and `calloc` functions and therefore more appropriate for real-time. VLPLAY provides a class for carrying out MIDI-driven musical synthesis when the synthesis methods for each program are fixed, i.e. without any meta-control. PLPLAY essentially inherits the VLPLAY classes and extends them to accommodate automated meta-control as well.

As described in Section 8.2.7, the VaPoR library also comes along with a set of VaPoR test programs and demos. VTCLOCK measures the time stamp accuracy for the current platform and is therefore very useful for porting VMCLOCK. VTGRAPH demonstrates basic window and keyboard/mouse input, and is therefore very useful for porting the CONSOLEQ class in VMSYSTEM and the GUI window support in VMGRAPHIC. VTRIFF demonstrates the panning and playing of arbitrary RIFF files, and is therefore useful for porting VMSOUND. VTWAVES and VTPROGS enable us to respectively load waveform and program libraries and review them. VTALLOC enables us to test the dynamic memory management of VLALLOC and compare its performance with that of the standard malloc and calloc services. VTPLAY lets us pan, play, record, and replay arbitrary MIDI files, and VTDEMO is an integrated program designed to demonstrate all of VaPoR's current combined capabilities.

B.5 Object-Oriented Programming

All VaPoR and PReTSL libraries were implemented in C++ to exploit its object-oriented capabilities. As demonstrated by the TIMER, CONSOLEQ, WINDOW, and SOUND classes, class abstractions enable us to create generic chunks of code and data which cater to all aspects of our run-time environment. Well-designed classes are readily incorporated into new applications, thereby significantly increasing software reuse and reducing time-to-market. When class capabilities must be extended, we can do so by creating derived classes which automatically *inherit* the capabilities of existing classes. When class capabilities must be customized, this can be accomplished via *virtual* methods. Both of these methods avoid code duplication and ease code maintenance. In the VLNODES module, for example, we have a basic NODE class for all synthesis nodes with several traits and methods which are common to all types of synthesis nodes, e.g. x, y coordinates, a symbolic name, a sample rate, a volume and an output buffer. All derived OSC, ENV, MIX, and AMP classes automatically inherit the internal data structures and methods provided by the NODE class. NODE also has several virtual methods such as init for allocating parameter vectors when each voice may have an independent MIDI-dictated control value, and go for carrying out the node synthesis of that node for a given voice. Such methods must be virtual so that each of the derived classes can customize these methods to fit its specific needs. As far as the real-time scheduler is concerned, all it sees is a list of synthesis nodes which can be activated in a uniform manner via the go method. Thus, we can create inheritance hierarchies in which each class can be dealt with at the highest possible level of abstraction with the same set of methods.

Another very useful C++ feature is support for dynamic binding. All available meta-control policies are derived classes of a basic METACTRL class, so they all have a common interface. The same applies to all clustering policies. Appropriate meta-control and clustering objects are created dynamically in accordance with the current needs of the application, whereupon a specific policy is called upon for a given decision epoch by merely updating a designated pointer.

Also used extensively is the ability to *overload* names, i.e. to let several class methods share a common name. The specific method to be used is identified by the number and nature of the parameters being used when the method is called. Methods with common objectives can thus share the same name, making the program much easier to follow and understand.

True, virtual methods and dynamic binding add an additional level of indirection when calling upon a class method. *Inline* (macro) methods, on the other hand, can reduce the level of indirection. All in all, we believe that, carefully used, object-oriented benefits can greatly outweigh any execution time penalties which may have been incurred by using the object-oriented technology. Once again, a primary top-layer advantage is that its source-level portability enables us to simply port to a more powerful platform if the subsequent performance is less than expected.