Safely Exploiting Multithreaded Processors to Tolerate Memory Latency in Real-Time Systems

Ali El-Haj-Mahmoud and Eric Rotenberg
Center for Embedded Systems Research
North Carolina State University, Raleigh, NC 27695-7256
(919) 513-2822

{aaelhaj,ericro}@ncsu.edu

ABSTRACT

A coarse-grain multithreaded processor can effectively hide long memory latencies by quickly switching to an alternate task when the active task issues a memory request, improving overall throughput. However, dynamic switching cannot be safely exploited to improve throughput in hard-real-time embedded systems. The schedulability of a task-set (guaranteeing all tasks meet deadlines) must be determined *a priori* using offline schedulability tests. Any computation/memory overlap must be statically accounted for. We develop a novel analytical framework that bounds the overlap between computation of a pipeline-resident-task and on-going memory transfers of other tasks. A simple closed-form schedulability test is derived, that only depends on the aggregate computation (C) and memory (M) components of tasks. Namely, the technique does not require specificity regarding the location of memory transfers within and among tasks and avoids searching all task permutations for a specific feasible schedule. To the best of our knowledge, this is the first work to provide the necessary formalism for safely and tractably exploiting coarse-grain multithreaded processors to tolerate memory latency in hard-real-time systems, exceeding the schedulability limits of classic real-time theory for uniprocessors. Our techniques make it possible to capitalize on higher frequency embedded processors, despite the widening processor-memory speed gap. Experiments with task-sets from C-lab benchmarks reveal improvement in the schedulability of task-sets, measured as the ability to schedule previously infeasible task-sets or reduce utilization for already feasible task-sets. We also demonstrate proof-of-concept by deploying our method in a cycle-level simulator of an ARM11-like embedded microprocessor augmented with multiple register contexts, the same hardware multithreading support available in Ubicom's IP3023 embedded microprocessor.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: *real-time and embedded systems*; C.1.3 [**Processor Architectures**]: Other Architecture Styles — *pipeline processors*.

CASES'04, September 22–25, 2004, Washington, DC, USA.

General Terms

Design, Performance.

Keywords

Multithreading, memory latency, real-time systems, schedulability test, worst-case execution time.

1. INTRODUCTION

Embedded microprocessors, like their desktop counterparts, are continuously evolving to achieve higher performance targets. For example, currently, the ARM11 embedded processor in a 0.13μ process can be clocked as high as 500 MHz and is expected to reach 1 GHz in a 0.10μ process [3]. At this rate, the processor-memory speed gap that is so noticeable in high-performance desktop/laptop computers will resurface in embedded systems.

Multithreaded processors [1][9][20][22][23][25] provide a means to scale system performance despite a growing processor-memory speed gap, by overlapping memory accesses of stalled threads with computation of other threads. Moreover, multithreading is especially pertinent to real-time embedded systems because these systems are characterized by task-sets with multiple periodic tasks, thereby providing thread-level parallelism and the opportunity to exploit multithreading even beyond what many desktop systems can [10].

There are many different forms of multithreaded processors, distinguished by their flexibility and granularity in overlapping instructions from multiple threads [25]. In this paper, we focus on hiding memory access latency, so we consider a coarse-grain form of multithreading whereby only a single thread uses the pipeline at a time and the current thread relinquishes the pipeline to another thread when it performs a memory access (e.g., [22]), sometimes called *switch-on-event blocked multithreading* [25], where the event is a memory access. The typical hardware support in this case is multiple register contexts for quickly switching the pipeline among threads. Ubicom's IP3023 processor [24], a scalar in-order embedded processor introduced in 2003, provides such support (8 register contexts). Although the IP3023 can also interleave instructions from multiple threads on a cycle-by-cycle basis, we do not explore this aspect in this paper (we consider only zero-cycle context-switching capability).

Multithreaded processors dynamically exploit computation/ memory overlap among tasks when overlap opportunities arise, which results in higher performance on average. However, no

guarantee can be given regarding the worst-case performance of the system. This is unsafe and unacceptable in a hard real-time embedded system, where all tasks must be statically guaranteed to meet their respective deadlines [5][11][17], by performing an offline schedulability test. Thus, any potential overlap must be analytically bounded and accounted for statically.

Statically bounding computation/memory overlap is not easy for a multithreaded processor that dynamically switches on memory accesses. First, the exact positioning of memory transfers within tasks must be known *a priori*. Then, all permutations of all tasks in a task-set must be examined for possible overlap opportunity. This exhaustive search may reveal a safe static schedule that can then be used afterwards at run-time. This approach is impractical and, most likely, intractable. Instead, a simple closed-form mathematical test is desired, that does not require searching for a specific schedule.

The crux of the problem is that switching only when actual memory accesses occur does not totally decouple otherwise independent threads. In fact, it introduces complex dependences among tasks from the standpoint of schedulability analysis, i.e., deducing when alternate tasks will be switched to. In addition to making analysis intractable, these false scheduling dependences needlessly defer future memory accesses of otherwise independent threads, squandering overlap opportunities.

Thus, the key lies in totally decoupling independent threads and this can be achieved by switching threads at frequent and regular intervals, via weighted-round-robin (WRR) scheduling [17]. WRR provides the needed scheduling policy on top of which we can build a novel analytical framework that safely and tractably models computation/memory overlap among multiple tasks, exceeding the schedulability limits of classic real-time theory for uniprocessors. In WRR, a task is resumed and then preempted once every round. We set the round equal to the memory latency. If a memory access is initiated before the preemption, then a precisely determinable fraction of the memory access is hidden with other tasks' computation. Moreover, if a memory access is not initiated before the preemption, system performance is no worse for it: assuming zero-overhead context-switching, controlled disruptions of a task do not affect its aggregate utilization of the processor, maintaining overall schedulability of the task-set.

Prior work with respect to WRR falls into three categories. (1) Fine-grain WRR has been applied in a non-real-time context (e.g., the HEP machine [20]), thus no worst-case schedulability formalism is developed. (2) The guaranteed-percentage policy has been applied to real-time scheduling, but only in the sense of conventional context-switching such that there is no attempt to overlap tasks' executions [17]. (3) The guaranteed-percentage policy and other classic policies have been evaluated in a multithreaded processor, but no formalism is provided to safely and tractably bound computation/memory overlap, i.e., no worst-case schedulability formalism is developed [7][15].

To the best of our knowledge, this is the first work to provide the necessary formalism for safely/tractably exploiting a coarse-grain multithreaded processor to tolerate memory latency in hard-real-time systems. Key facets of this work include:

(1) For task-sets with a memory time component, our framework provides a safe and tractable means for exceeding the schedulability limits of classic real-time theory for uniprocessors. This means task-sets that were previously unschedulable may become schedulable, or, for already-schedulable task-sets, new tasks can be added and/or rates increased. Moreover, for task-sets with little or no memory time, schedulability is not lessened.

(2) We derive a closed-form schedulability test. This means a very simple mathematical test can be used to determine whether a hard-real-time task-set is schedulable, in the context of WRR on a multithreaded processor. The test is based only on the periods and worst-case execution times (WCET) of tasks. Note that these most basic task parameters are the basis for all classic real-time scheduling theory, and are therefore already available. This also means that our analytical framework is compatible with real-time system design environments.

(3) Our framework does not require any knowledge of where memory accesses occur within and among tasks. We only need to know the worst-case number of memory accesses in each task, which is already available as a byproduct of the separate and orthogonal worst-case execution time (WCET) analysis phase [2][12][18] that underpins schedulability analysis for all hard real-time systems. Moreover, our technique is independent of the type of level-1 memory structure – hardware-managed cache vs. software-managed scratchpad memory – since the conventional WCET analysis phase employs the necessary methods for bounding the worst-case number of memory accesses for either structure [12].

(4) Our analytical framework accounts for practical memory system issues, such as the degree of parallelism in the memory system (memory banks) and serialization on the bus.

To sum up, our approach provides a path towards capitalizing on higher frequency processors in the real-time embedded systems domain, in spite of lagging memory speeds. Our framework is safe in that it statically bounds the amount of overlap among tasks under all possible scenarios. In addition, it is tractable, in that worst-case schedulability can be confirmed/disconfirmed with a closed-form schedulability test that we derive. This test is based only on the memory-to-computation ratio of each task individually, without having to consider the exact positioning of memory requests within and among tasks.

The rest of this paper is organized as follows. In Section 2, we give a brief background on classic real-time scheduling, which does not take into consideration any execution overlap. In Section 3, we discuss the issues involved in boundedly and tractably overlapping memory latency on a multithreaded processor. In Section 4, we derive a closed-form schedulability test in the context of weighted-round-robin, and discuss assigning safe weights to tasks to guarantee meeting deadlines. Section 5 outlines our experimental methodology, chiefly characterizing our task-sets and describing our simulation infrastructure. Section 6 presents schedulability experiments for the task-sets, augmented with simulation experiments for demonstration. Section 7 reviews related work and we summarize in Section 8.

2. BACKGROUND ON CLASSIC REAL-TIME THEORY FOR UNIPROCESSORS

A hard-real-time embedded system is characterized by a collection of recurring tasks, called a task-set, and a key goal is to determine *a priori* whether or not the task-set is schedulable as a whole. Each task is characterized by three parameters that are needed to determine schedulability:

1. *Period:* New instances of the task are "released" – made available for execution – at regular time intervals equal to the period of the task. For example, in Figure 1, the periods of tasks A and B are 8 and 4 time units, respectively.

2. *Deadline:* This is the time by which an instance of the task must complete. For tractable schedulability analysis [17], the deadline is often set equal to the period, meaning a task instance must complete before the next instance is released, as shown in Figure 1 for tasks A and B.

3. *Worst-case execution time (WCET):* The WCET is an upper bound on the execution time of an instance of the task, and is guaranteed never to be exceeded. For example, in Figure 1, the WCETs of tasks A and B are 2 and 3 time units, respectively.

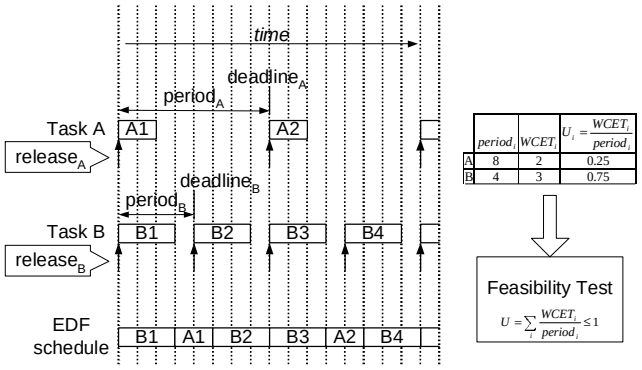

Figure 1. Example task-set composed of two periodic hard-real-time tasks.

Interestingly, for well-studied scheduling policies, the schedulability of a task-set can be determined *a priori* using offline tests that require only knowledge of tasks' periods and WCETs. For example, Liu and Layland [16] derived classic results for the earliest-deadline-first (EDF) and other uniprocessor scheduling policies. Most interestingly, they derived a very simple test for determining whether or not a task-set is schedulable using the EDF policy, namely that the sum of all tasks' utilizations must be less than or equal to 1 (indicating that the processor is not over-subscribed). The utilization of a task is the fraction of time that the processor spends executing the task. In the worst case, the utilization of a task i is its WCET divided by its period: $U_i = \frac{WCET_i}{period_i}$. Thus, the schedulability test is simply: $U_{total} = \sum_i \frac{WCET_i}{period_i} \leq 1$. The example task-set in Figure 1 is schedulable because the sum of the tasks' utilizations is 0.25 + 0.75 = 1. At run-time, the real-time operating system (RTOS) will dynamically prioritize currently-released tasks based on earliest deadlines, producing the schedule at the bottom of Figure 1.

A key basis for the Liu and Layland result and the whole of classic real-time scheduling theory is treating the processor as a single indivisible resource. Although the processor is shared among tasks and tasks may preempt one another (context-switching), it is assumed that only one task has possession of the *entire* processor at a time. Using this model, memory access latency from one task cannot be overlapped with computation from another task. Essentially, this assumption treats a multithreaded processor as a *non*-multithreaded one, missing the opportunity for bridging the processor-memory speed gap in the real-time domain.

A formalism that accounts for overlap of tasks' WCETs can effectively reduce perceived worst-case utilization, and thereby make task-sets more schedulable. Task-sets that previously were not schedulable may become schedulable, more tasks can be added to already-schedulable task-sets, or periods of tasks can be reduced (rates increased) in already-schedulable task-sets.

3. MOTIVATION FOR DETERMINISTIC SWITCHING

In this section, we illustrate the complexity of bounding overlap on a multithreaded processor that uses a dynamic switch-on-event approach for tolerating memory latency. We then describe how a deterministic switching policy decouples independent threads, providing a foundation for developing a tractable analytical framework.

3.1 Intractability of Dynamic Switching

Figure 2(a) shows a task-set composed of two tasks, A and B, each with utilizations of 1 ($WCET_A$=$period_A$, $WCET_B$=$period_B$). The EDF schedulability test for this task-set fails on a uniprocessor system (worst-case utilization = 2, which is greater than the uniprocessor utilization limit of 1). However, this task-set can be scheduled on a multithreaded processor if memory accesses from one task are overlapped with pipeline computation from the other task, and *vice versa*. Figure 2(b) shows such an example. In the example, the processor supports multiple pending memory requests, i.e., it has multiple *memory transfer units* (MTU) if using a software-managed scratchpad memory, or multiple *miss status handling registers* (MSHR) if using a hardware-managed cache. Tasks A and B each have two memory transfers ("m"), interleaved with computation as shown in Figure 2(b). Each task has access to a private memory transfer unit (A uses MTU1, B uses MTU2), and the processor dynamically switches the active thread when a memory access is performed. A feasible schedule can be found whereby all of task A's memory component overlaps with task B's computation component, and *vice versa*. In this way, the task-set is schedulable using this uniprocessor.

However, suppose we replace task B with task B', as shown in Figure 2(c): they have the same WCET breakdown in terms of computation versus memory time, but the memory and computation components are interleaved differently in the new task B'. Whereas the task-set {A, B} is schedulable, task-set {A,

B'} is not (B' misses its deadline as shown in the figure), even though tasks B and B' have the same memory-to-computation ratio.

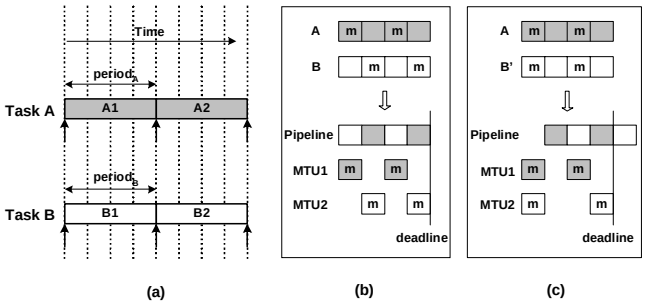

Figure 2. Overlapping execution of different task-sets on a multithreaded processor. (a) Task-set composed of tasks A and B ($WCET_A$=$period_A$=$WCET_B$=$period_B$). (b) A and B with two memory transfers each ("m") are schedulable because of computation-memory overlap. (c) B' is identical to B except for positioning of memory transfers, and is not schedulable with A.

We conclude that a simple utilization-based schedulability test – like the EDF test – is not enough to determine schedulability of a task-set on a multithreaded processor using a dynamic switch-on-event policy. Such a schedulability test would only consider the relative proportions of memory and computation time in each task's WCET, without regard for any specific positioning of computation and memory among tasks. Yet, the interleaving of memory and computation must also be considered, in which case there is no closed-form schedulability test but rather an explicit and potentially exhaustive search for a valid schedule.

3.2 Tractability through Deterministic Switching

The previous subsection underscores the complexity of bounding memory overlap using dynamic context-switching. We can tractably bound the amount of overlap between memory time of a task and computation time of other tasks by forcibly creating overlap opportunities. This is achieved by forcing task switches in a repeating *weighted-round-robin* (WRR) sequence. A *round* is a fixed time interval during which each task is given a single time-slot for execution on the pipeline. Thus, each task has possession of the pipeline for a certain fraction of each round – the task's *duty cycle* – as shown in Figure 3 for four tasks (T1-T4).

By enforcing duty cycles, we dilate the WCETs of all tasks (since tasks are forcibly preempted), but this is offset by the fact that a guaranteed duty cycle effectively gives each task its own, private pipeline (or *virtual processor*), as shown in Figure 3. We simply need to ensure that (1) the duty cycle of each task is sufficient to complete the dilated task before its deadline on its virtual processor (i.e., satisfy the condition: dilated WCET ≤ period), and (2) the sum of all tasks' duty cycles is less than or equal to 1, tying virtual processors back to the physical processor from which they derive. In fact, these two conditions are the schedulability test.

Now, we just need to determine the amount by which each task's WCET is dilated based on its duty cycle, for evaluating condition (1) above. A forced preemption can occur during computation or during a memory transfer. If it happens during computation, WCET is dilated because the task becomes completely idle, doing neither computation nor a memory transfer. This scenario is highlighted in Figure 3 for the first forced preemption of T4. However, if a task manages to initiate a memory transfer before being forcibly preempted, the transfer will continue in spite of the forced preemption, thanks to the task's private MTU. The key to our approach is to set the round equal to the latency of a memory transfer. This ensures that a memory transfer, regardless of where it occurs within a task, will begin and end in consecutive duty cycles of the task, as shown for rounds i+1 and i+2 of T4 in Figure 3. In this way, WCET is not dilated by forced preemptions during memory transfers, since finishing a memory transfer is marked by immediate resumption of computation. Moreover, this result holds independent of where memory transfers occur within the task. This is significant because it means we can mathematically model a task as being composed of two separable time components, total computation time C and total memory time M, where C is dilated by forced preemptions but M is not. And, it is easy to determine the factor by which C is dilated: C is dilated by the inverse of the duty cycle (e.g., if duty cycle = 0.5, computation time doubles). Thus, WCET simply expands from [C + M] to [C/d + M], where d is the duty cycle.

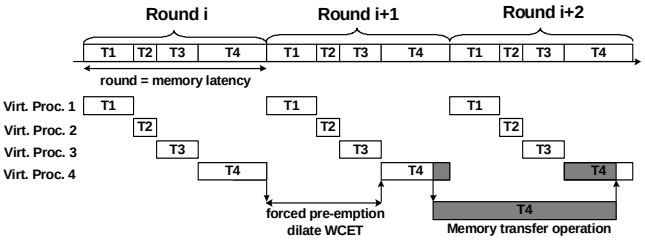

Figure 3. Exploiting WRR scheduling for bounding overlap.

A simple closed-form schedulability test results. First, we ensure schedulability of each task on its own virtual processor by selecting the minimum duty cycle such that its dilated WCET is less than its period (deadline), i.e., select minimum d (where d ≤ 1) such that [C/d + M] ≤ period. Second, we evaluate overall schedulability on the physical processor by checking whether or not the sum of all tasks' duty cycles is less than or equal to 1.

4. ANALYTICAL FRAMEWORK

For safety guarantees in hard-real-time systems, we must statically bound the amount of possible overlap among tasks. At the same time, we want to avoid exhaustively searching potential scenarios for proving/disproving worst-case schedulability of task-sets. In this section, we derive closed-form schedulability tests for our platform (based on the WRR scheduling policy as described in Section 3.2) to achieve the two goals of safety and tractability.

Our analysis begins with the specific case of a single task per virtual processor (Section 4.1). We then generalize this result to allow for multiple tasks per virtual processor (Section 4.2), enabling the use of task-sets with arbitrary numbers of tasks.

Note that our analysis is with respect to the number of *virtual processors*. This provides an abstraction of the underlying hardware that in no way places constraints on either the processor or memory system design. Rather, the opposite is true, i.e., the underlying processor plus memory system implementation dictates the number of available virtual processors and the analytical model is configured accordingly. In high-level terms, the number of virtual processors reflects the overall thread-level and memory-level parallelism in the system. Specifically, the number of virtual processors is the minimum of (1) the number of register contexts, (2) the number of pending memory requests (i.e., number of parallel MTUs or MSHRs), and (3) the number of DRAM banks (for parallel DRAM accesses). In Section 4.3.2, we extend the analytical framework to decouple the number of virtual processors from the number of parallel DRAM banks, so that we can capitalize on some overlap opportunity even with limited parallelism in the DRAM. In Section 4.3.1, we describe how to safely account for serialization of transfers on the memory bus.

4.1 Single Task per Virtual Processor

With WRR, each task is allocated a fixed time-slot in each round. The fraction of each round allocated to a task is called the task's *duty cycle*, d, where $0 < d \leq 1$. Recall, in Figure 3 of Section 3.2, we showed an instance of task T4 executing on the pipeline during its duty cycles. Notice, we set the round equal to the memory transfer latency, ensuring that individual memory transfers begin and end in consecutive duty cycles.

We divide the task's WCET into computation (C) and memory (M) components. The number of *whole* rounds (i.e., assuming no disruptions by memory transfers in the middle of duty cycles) needed to complete the computation component of a task is its computation time divided by the time per round allocated to the task, or $N = \lceil C/(d \times R) \rceil$, where C is aggregate computation time, d is the duty cycle, R is the round time, and N is the number of whole rounds. This expression holds in spite of disruptions by memory transfers and is independent of when these disruptions occur. When computation is disrupted during a duty cycle by a memory transfer, computation resumes at the corresponding point in the next duty cycle, as shown in Figure 3 in Section 3.2. Since we separate out memory time explicitly, the effect is to concatenate complementary computation portions of adjacent duty cycles, as if the disruptions had not occurred.

The time needed to finish the computation component is the number of whole rounds multiplied by the round time, or $N \times R = \lceil C/(d \times R) \rceil \times R$. Since individual memory transfers always begin and end in consecutive duty cycles, we ensure that there is no idle time following transfers. Therefore, aggregate memory time M is not dilated. Thus, we get the following expression for WCET', the dilated WCET: $WCET' = \lceil C/(d \times R) \rceil \times R + M$.

The ceiling function $\lceil \ \rceil$ is a necessary precaution. An interval of time equal to the round is guaranteed to contain one full duty cycle (in aggregate), regardless of where the interval starts and ends. The ceiling function produces an integer number of rounds, N, guaranteeing N duty cycles regardless of where the task is released.

Providing each task with a fixed time-slot on the pipeline (for computation) and a dedicated MTU (for memory transfers) is like assigning each task to a *virtual processor* (VP), for which there is no contention. Thus, assuming one task per virtual processor, the only constraint within a virtual processor is that the task's dilated WCET must be less than or equal to its period, so that the current instance of the task finishes before the next instance is released:

Equation 1. $\lceil C/(d \times R) \rceil \times R + M \leq period$

It turns out that, if we constrain the period to be an integer multiple of the round, then we can correctly remove the ceiling function from the left-hand side of Equation 1 (this is confirmed, below). We do not sacrifice system timing specifications if we replace the period in Equation 1 with a tighter period that is an integer multiple of the round, i.e., $period' = \lfloor period/R \rfloor \times R$. If we remove the ceiling function from Equation 1, replace the period with period' (tighter constraint), and solve for d, we get $d \geq C/(period' - M)$. Since we want to minimize the duty cycle (i.e., minimize utilization of the physical pipeline by this task), we solve for d as follows:

Equation 2. $d = C/(period' - M)$

We now substitute this d back into Equation 1 to confirm that initially removing the ceiling function is correct, assuming the modified period. This exercise yields the following: $\lceil (period' - M)/R \rceil \leq (period' - M)/R$. This condition only holds if both period' and M are integer multiples of R, which is the case: (1) the round R is equal to the memory latency, and M is an integer multiple of the memory latency; (2) we defined period' to be an integer multiple of the round R.

Equation 2 ensures schedulability of individual tasks on their virtual processors. We determine overall schedulability of the task-set as a whole by checking whether or not the sum of all tasks' duty cycles is less than or equal to 1.

Equation 3. $\sum_i d_i \leq 1$

The impact of using period' versus period is minor because the round is typically a small fraction of the period. For example, the WCET of our smallest task (lms) is 0.16 ms, or 160,000 cycles at 1 GHz. Even with the tightest possible period of 0.16 ms, and a round of 100 cycles (memory latency), the round is less than 1/1000 the period.

A more significant effect (but still relatively small) is rounding up $d \times R$ to be an integer number of cycles of the round, during which the task is active. Duty cycle rounding is only a problem for a task-set that is barely feasible using conventional EDF

scheduling and that does not have a perceptible memory component. With no memory to overlap, duty cycle rounding is enough to make the task-set barely infeasible using adapted WRR. In this case, we can simply revert to using conventional EDF scheduling.

4.2 Multiple Tasks per Virtual Processor

The analysis in the previous subsection assumes there is one real-time task per virtual processor. Now, we extend the results to the more general case of supporting more than one task per virtual processor (e.g., when the number of hardware contexts is less than the number of tasks in a task-set).

When there are multiple tasks on a single virtual processor, their WCETs cannot be overlapped because there is only one register context, one memory transfer unit, etc. That is, a virtual processor is logically a conventional single-threaded uniprocessor. As such, conventional uniprocessor scheduling is required within the virtual processor – we use conventional EDF.

The duty cycle expression in Equation 2 is generalized by realizing that a duty cycle d is associated with a virtual processor, not any particular task. WCETs of all tasks on a virtual processor are dilated by that virtual processor's duty cycle d. Only their computation components are dilated, yielding the following condition for EDF schedulability of t tasks on a single virtual processor.

Equation 4.
$$\frac{\left(\frac{C_1}{d}+M_1\right)}{P_1}+\frac{\left(\frac{C_2}{d}+M_2\right)}{P_2}+\cdots+\frac{\left(\frac{C_t}{d}+M_t\right)}{P_t}\le 1$$

Each term in the above expression is the modified (perceived) utilization of a single task, i.e., dilated WCET divided by period (P), and EDF schedulability is assured if the sum of all tasks' modified utilizations is less than or equal to 1. Using exactly 1 will minimize d. Equation 4 can be simplified as follows.

Equation 5.
$$\left(\frac{C_1/P_1}{d}+M_1/P_1\right)+\left(\frac{C_2/P_2}{d}+M_2/P_2\right)+\cdots+\left(\frac{C_t/P_t}{d}+M_t/P_t\right)=1$$

Solving for d yields the generalized result below.

Equation 6.
$$d=\frac{\sum_{j=1}^{t}\frac{C_j}{P_j}}{1-\sum_{j=1}^{t}\frac{M_j}{P_j}}$$

The schedulability test of Equation 3 still applies. Note that the specialized Equation 2 is consistent with the general form Equation 6 for t=1.

4.3 Modeling the Memory System

4.3.1 Modeling Bus Transfer Time

So far, we have separated WCET into computation (C) and memory (M) components. M only accounts for the raw DRAM access time. However, a bus transfer accompanies every DRAM access, which is not accounted for by either the C or M components. We introduce another WCET component, B, to reflect aggregate bus time of a task: the total time spent by a task transferring its memory blocks to/from DRAM.

Bus transfer requests from multiple virtual processors are serialized on the memory bus. In the worst case, a virtual processor may have to wait for (n-1) other transfers to complete before it can own the bus, one for each of the other virtual processors (assuming there are n virtual processors). Thus, in the worst-case, a transfer takes n times as long to complete. Whereas the aggregate bus time for the baseline system is B, the aggregate bus time for the system that exploits overlap is n*B. Thus, a tradeoff is revealed: the aggregate bus time of a task is extended (n*B) but we can overlap aggregate memory time plus bus time of the task with computation of other tasks.

The dilated WCET in this case will be $WCET' = (C/d) + (M + n \times B)$ and the duty cycle $d = C/(period' - M - n \times B)$.

Using this model, the round is set equal to the latency of one DRAM access plus the extended (times n) bus transfer time of one memory block.

Note that the base case does not suffer from this worst-case extension of bus time. Bus conflicts among different tasks cannot occur because all tasks are serialized anyway (no overlap of tasks' WCETs).

4.3.2 Modeling Memory Banks

We first consider the case where the number of virtual processors is equal to the number of DRAM banks. We prevent bank conflicts from occurring by mapping virtual processors to DRAM banks, one-to-one. For example, virtual processor 1 is mapped to bank 1, meaning any tasks that run on virtual processor 1 have their instructions/data allocated to bank 1. Tasks on the same virtual processor are serialized on that virtual processor; hence allocating them to the same bank does not introduce conflicts. Tasks on different virtual processors are prevented from conflicting by ensuring their instructions/data are allocated to different banks, corresponding to the virtual processors. Thus, DRAM parallelism is fully exploited.

Next, we extend our analysis to decouple the number of virtual processors from the number of DRAM banks. Thus, the number of virtual processors is governed only by characteristics of the processor core (namely, number of register contexts and MTUs/MSHRs).

If the number of DRAM banks is less than the number of virtual processors, then multiple virtual processors share the same DRAM bank and conflicts may occur. In this case, the memory access latency from the perspective of a virtual processor is extended, in the worst case, by a factor s, where s is the number of virtual processors sharing a single bank. Each access from the virtual processor assumes that the bank is already busy, and has to wait for (s-1) other accesses, in the worst-case, to finish before it can proceed. The total memory component M is thus extended to s*M.

We can now express the dilated WCET as $WCET' = (C/d) + (s \times M + n \times B)$ and the duty cycle as $d = C/(period' - s \times M - n \times B)$. As with bus conflicts, bank conflicts reveal a tradeoff: the aggregate memory time is extended to s*M but we can overlap it with computation of other tasks.

Using this model, the round is set equal to the extended (times s) DRAM access latency plus the extended (times n) bus transfer time of one memory block.

Note that the base case does not suffer from this worst-case memory latency extension. Bank conflicts among different tasks cannot occur because all tasks are serialized anyway (no overlap of tasks' WCETs).

5. EXPERIMENTAL METHODOLOGY

The primary experiments do not involve simulation since schedulability analysis is based solely on schedulability tests for the baseline EDF and our adapted WRR. WCETs and periods of tasks are inputs to these schedulability tests.

Static worst-case timing analysis is used to derive a task's WCET on a particular microarchitecture. Thus, we first summarize the microarchitecture in Section 5.1. Static worst-case timing analysis is covered briefly in Section 5.2. We then characterize the tasks and task-sets used in our analytical experiments, in Section 5.3.

We also implemented a detailed cycle-level simulator of the microarchitecture as a run-time demonstration vehicle. The simulator is covered in Section 5.4.

5.1 Microarchitecture

Table 1 summarizes the microarchitecture model assumed for both deriving WCETs and performing simulations. The processor core is modeled after the ARM11 scalar in-order 8-stage pipeline [3], with slight modifications to support the Simplescalar PISA ISA [4]. The processor provides four register contexts. Each thread also has a special-purpose register containing the number of cycles the thread is active during each round. WRR scheduling is performed by hardware: when a thread is switched to, a counter is loaded with the contents of that thread's cycle-count register. The counter counts down, and, when it reaches zero, the next thread in line is scheduled.

Table 1. Microarchitecture configuration.

Processor Core	Simplescalar PISA ISA
	Scalar, in-order, 8-stage pipeline (*ARM11*)
	Static (BT/FNT) branch prediction Misprediction penalty = 6 cycles
	4 register contexts
	4 memory transfer units (MTU)
Core Latencies	Address generation = 1 cycle
	Integer ALU ops = 3 cycles (*ARM11*)
	Complex ops = MIPS R10K latencies
Level-1 Scratchpad Memories (on-chip)	(*based on Ubicom IP3023*) Instruction scratchpad: 256KB Data scratchpad: 64KB Block size: 128 bytes Access time: 2 cycles
Memory System	# banks: default = 4, also varied (1, 2, & 4)
	DRAM access time = 50 ns/block
	Bus transfer time = 64 ns/block

When the current task is preempted to make way for the next task, the next task must proceed unobstructed because schedulability analysis assumes a guaranteed duty cycle for each task. However, the instruction in the issue stage of the pipeline may be stalled waiting for a long-latency instruction (e.g., floating-point arithmetic) to produce one of its source operands. One solution is to squash the instructions that are in the front-end stages of the pipeline (4 instructions in the case of ARM-11) during each task switch, so that the next task may proceed unobstructed. However, this introduces a 4-cycle penalty for resuming tasks (to re-fetch and re-decode the squashed instructions). If there are four tasks and only, say, 50 cycles in a round, then resuming tasks consumes 16 cycles out of the 50-cycle round. Instead of squashing instructions in the front-end of the pipeline, we augment the front-end of the pipeline with four shadow latches per pipeline latch (one shadow latch per virtual processor). When the current task is preempted, its front-end pipeline state is checkpointed by one set of the shadow latches. Instructions from the next task proceed through the front-end of the pipeline unobstructed. When the preempted task is resumed, it copies its saved state from the shadow latches back into the pipeline latches, continuing execution from exactly where it had left off.

The memory hierarchy consists of two level-1 scratchpad memories modeled after Ubicom's IP3023 [24], a 256KB I-scratchpad for instructions and a 64KB D-scratchpad for data, backed by off-chip DRAM. Scratchpad memories are essentially software-managed caches. The ISA is augmented with three types of memory transfer instructions: fetch instruction block (retrieve a block from off-chip memory to the I-scratchpad), fetch data block (retrieve a block from off-chip memory to the D-scratchpad), and flush data block (write-back a block from the D-scratchpad to off-chip memory). Memory transfer instructions specify an off-chip block address and a scratchpad block address. Blocks are 128 bytes.

Memory transfer instructions are executed by memory transfer units (MTU). There are four MTUs. As explained in Section 4, the combination of four register contexts and four MTUs provides four available virtual processors.

A memory transfer consists of two phases, the DRAM access and the memory bus transfer (transferring bytes on the bus). Latencies for each are given in Table 1. The default DRAM has four banks that can be accessed in parallel, although we also perform experiments varying the number of DRAM banks (1, 2, and 4 banks).

The I-scratchpad and D-scratchpad are statically partitioned among tasks in the task-set. Cache partitioning (or scratchpad partitioning, in our case) is a commonly used technique for eliminating conflicts among tasks [14][19][26], simplifying static worst-case timing analysis. In short, partitioning is needed for safe real-time scheduling regardless of the policy (baseline EDF or our adapted WRR).

Memory transfer instructions are manually inserted in the tasks (by the programmer) to fetch instruction/data blocks from off-chip memory to the scratchpads before they are accessed by the instruction fetch unit and by loads/stores, ensuring these

references always "hit". Dirty data blocks that will be re-referenced later are explicitly written back to main memory when they need to be displaced to make room for new blocks.

5.2 Static Worst-Case Timing Analysis

Although we have access to static worst-case timing analysis tools capable of bounding WCETs of hard-real-time tasks on simple scalar pipelines, it is beyond the scope of this paper (and orthogonal to it) to port one of these tools to model the microarchitecture used in this paper. Thus, we performed manual analysis assisted with simulation [18] to safely yet tightly bound tasks' WCETs.

Our manual analysis is procedurally similar to the bottom-up fixed-point approach described by others [12]. We find longest timing paths within inner loops and leaf functions, and work upwards towards outer loops and functions at higher levels. Forward branches are handled by selecting the longest of two timing paths, after padding the taken path with the misprediction penalty (6 cycles), since static branch prediction predicts forward branches as always not-taken. Backward branches are handled by padding the loop continuation with the misprediction penalty, since static branch prediction predicts backward branches as always taken. After manually identifying longest timing paths, we use simulation assistance to tightly model overlapped execution of instructions along these paths.

After bounding the computation time component (C) of WCET, we add on the memory time component (M) and the bus time component (B) based on the total number of programmatic memory transfers in the task.

We explicitly avoided placing memory transfer instructions in conditional paths (i.e., hammocks), to make aggregate computation time and aggregate memory time of WCET easily separable. If a memory transfer instruction forms one side of a hammock and computation the other side, timing analysis will include the hammock in either C or M depending on which side of the hammock takes more time. However, the two sides are affected differently by duty cycles – M is not dilated whereas C is. Thus, if we allow memory transfer instructions inside hammocks, WCET analysis may have to be modified. A simple solution is to always include hammocks in C, trading overlap opportunity for safe analysis. Another simple solution is to logically include the memory transfer in both sides, which has the safe effect of moving the transfer latency in series with and out of the hammock. Finally, worst-case timing analysis could be explicitly modified to work in tandem with the WRR scheduling, taking into account the duty cycle of a task (WCET parameterized in terms of duty cycle). In our experience with explicit management of scratchpads, we found it unnecessary and over-complicated to embed transfers in hammocks.

5.3 Task-sets

To compose the task-sets, we use five tasks from the C-Lab real-time benchmark suite [6], shown in Table 2. These benchmarks are extensively used in real-time research, because they explicitly avoid irregular programming constructs that complicate worst-case execution time analysis. The benchmarks are compiled to the Simplescalar ISA [4] with –O3 optimization enabled.

The second column of Table 2 shows the total WCET of each benchmark at a processor frequency of 1 GHz, derived by WCET analysis, assuming no overlap. The next three columns break down the components of WCET. C is the total computation time, B is total bus transfer time, and M is the total memory time. The sixth column shows the number of memory transfer instructions in each benchmark (total transfers, fetch data block into and flush data block from the D-scratchpad, and fetch instruction block into the I-scratchpad). The final column gives the average actual execution time of each benchmark with no other tasks running, measured on the cycle-level simulator. We constructed various task-sets with different memory utilizations by combining tasks from Table 2. Each task-set in Table 3 is composed of a single task per virtual processor (assuming a four virtual processor system), and characterized by its memory-to-computation ratio. Task-sets with comparatively high, moderate, and low memory-to-computation ratios are referred to as HIGH, MED, and LOW, respectively. Task periods are chosen to yield a fully utilized system (U=1) at 1 GHz using our proposed adapted WRR, i.e., $\sum_i d_i = 1$. This implies the task- sets are just-feasible using our technique. Thus, if the task-set has a perceptible memory component, it will not be feasible using conventional EDF scheduling. This setup allows us to measure the over-subscription of the EDF schedule, whether or not task-sets will become feasible using EDF if frequency is increased, the amount of static slack achieved by WRR over EDF, etc.

Table 3 lists the tasks in each task-set. The task's name, period (P), and individual utilization ($U_i = WCET_i/P_i$) are indicated for each task i (WCETs were provided in Table 2). The second-to-last column gives the contribution of memory (DRAM + bus) to worst-case utilization of each task-set (assuming no overlap), revealing the memory-intensiveness of each task-set, ranging from 0.331 for HIGH down to 0.0440 for LOW. The last column gives the total worst-case utilization of each task-set using EDF scheduling – none of the task-sets are provably schedulable because their worst-case utilizations are greater than 1, failing the EDF schedulability test.

We also constructed HIGH, MED, and LOW task-sets composed of eight tasks each, two tasks per virtual processor. The details of these task-sets are not shown here for space constraints, although we discuss their results in the next section.

5.4 Cycle-Level Simulation Environment for Run-Time Experiments

We implemented a detailed cycle-level simulator (custom-built using Simplescalar toolset [4]) that models the microarchitecture described in Section 5.1, as a run-time demonstration vehicle. EDF and adapted WRR scheduling use the same microarchitecture substrate.

A lightweight software EDF scheduler is used to schedule multiple tasks on the same virtual processor. WRR among the four virtual processors is implemented via four hardware registers, as described in Section 5.1.

Table 2. C-lab benchmark description (processor frequency = 1 GHz).

Task	WCET (ms)	WCET components			# memory transfer instr. (total / D-scratch. / I-scratch.)	Avg. exec. time (ms)
		C (ms)	B (ms)	M (ms)		
adpcm	3.35	3.29	0.0328	0.0256	512 / 490 / 22	2.45
cnt	0.170	0.120	0.0282	0.0221	441 / 425 / 16	0.160
mm	5.15	4.36	0.442	0.345	6908 / 6884 / 24	5.08
lms	0.159	0.154	0.00333	0.00260	53 / 37 / 16	0.155
srt	2.26	2.26	0.00256	0.00200	40 / 30 / 10	1.88

Table 3. Task-sets composed from C-lab benchmarks (Utilization of adapted WRR is 1, at 1GHz).

Task-set	VP 1			VP 2			VP 3			VP 4			U_{mem} (EDF)	U_{total} (EDF)
	Name	P (ms)	U	Name	P (ms)	U	Name	P (ms)	U	Name	P (ms)	U		
HIGH	cnt	0.620	0.274	cnt	0.620	0.274	Cnt	0.594	0.286	cnt	0.594	0.286	0.331	1.12
MED	mm	18.9	0.272	mm	18.9	0.272	Mm	20.4	0.252	mm	20.4	0.252	0.162	1.05
LOW	srt	11.4	0.198	lms	1.65	0.0963	cnt	1.98	0.0858	adpcm	5.32	0.629	0.0440	1.01

6. EXPERIMENTS

In this section, we present worst-case schedulability experiments for baseline EDF (no overlap of tasks' WCETs) and our adapted WRR (overlap of tasks' WCETs). We show results for a single task per virtual processor and multiple tasks per virtual processor. In addition, we study the effect of varying the number of DRAM banks, including evaluating fewer banks than the number of virtual processors to understand the impact of limited DRAM parallelism.

Finally, for demonstration, we simulate the baseline EDF and our adapted WRR on the cycle-level simulator for 100ms. In all cases, the simulation results are in agreement with the schedulability tests.

6.1 Schedulability Experiments

The graph in Figure 4 shows results of schedulability tests for a four virtual processor system with four DRAM banks. Each task-set has four tasks (thus, for WRR, there is a single task per VP). The first bar ("EDF") is the worst-case utilization under EDF scheduling, i.e., the sum of individual task utilizations, which must be less than 1 for schedulability. The next bar ("WRR") is the worst-case utilization using our proposed adapted WRR (taking into account overlapping WCETs), i.e., the sum of all tasks' duty cycles, which must be less than 1 for schedulability. Recall, we composed our task-sets to achieve a worst-case utilization of 1 using adapted WRR at 1 GHz, and this is evident from the graph.

We also show a third bar, labeled "perfect", which represents an ideal lower bound on worst-case utilization. To model ideal overlap of computation and memory time, we set M=0 and B=0 (hiding all DRAM latency and bus transfer time) in the tasks' WCETs and plotted worst-case utilization accordingly. Thus, the difference between the "EDF" and "perfect" bars is the memory component of worst-case utilization, including the bus transfer time (same as U_{mem} column of Table 3). The larger this gap, the more potential reward for WRR scheduling. This gap increases, going from least memory-intensive task-set (LOW) to most memory-intensive task-set (HIGH).

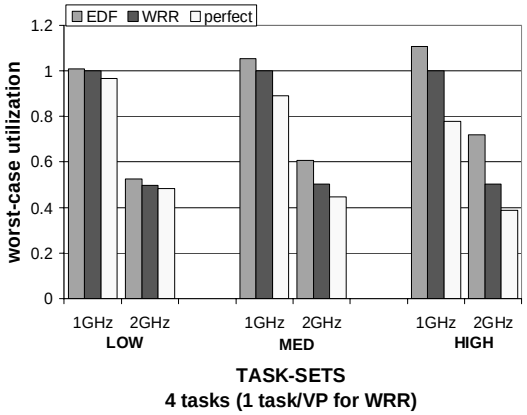

Figure 4. Worst-case utilization.

Task-sets LOW, MED, and HIGH are infeasible at 1 GHz using the conventional EDF (EDF worst-case utilization exceeds 1), whereas WRR exploits overlapping WCETs in an analytically-bounded way to produce a feasible schedule.

Even for feasible EDF scenarios at the higher frequency (2 GHz), using WRR results in more static slack in the schedule than does EDF, e.g., Figure 4 shows 50% slack for "WRR" vs. only 29% for "EDF", for HIGH at 2 GHz. Static slack can be used to increase functionality via adding more tasks, reducing periods, etc.

Notice that "WRR" approaches the "perfect" point, but does not perfectly overlap computation and memory time because memory transfer instructions initiate and complete in adjacent duty cycles, wasting an aggregate of one whole duty cycle during which the task could use the pipeline but does not. This is evident from the example memory transfer in Figure 3 (Section 3.2).

Figure 5 shows that our framework is scalable to systems where the number of tasks is greater than the number of available register contexts, by supporting multiple tasks per virtual processor. Conventional software EDF is used to schedule

multiple tasks within a virtual processor. The same observations discussed previously, for a single task per VP, still apply for this more general case.

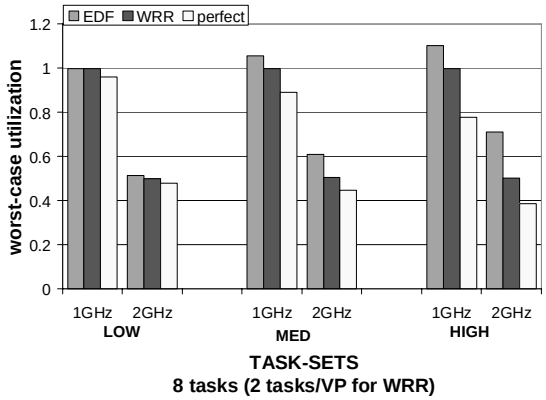

Figure 5. Worst-case utilization.

In Figure 6, we show the effect of varying the number of DRAM banks (i.e., DRAM parallelism) on the schedulability of WRR. "EDF" and "perfect" bars are the same as before, because they are not affected by the number of DRAM banks (no conflicts). "WRR-1", "WRR-2", and "WRR-4" bars present WRR schedulability results for a four virtual processor system, with 1, 2, and 4 total DRAM banks, respectively. The trend is that schedulability improves with more banks, as anticipated. Somewhat surprisingly, note that "WRR-1", which essentially serializes all memory accesses like "EDF", still performs better than "EDF" for the 2 GHz processors. Only at 1 GHz and 1 DRAM bank is the single-threaded EDF approach slightly preferred.

Although all memory accesses are essentially serialized in "WRR-1" due to our very conservative assumptions regarding bank and bus conflicts, they are still overlapped with pipeline computation from other tasks, unlike "EDF". Results are very positive for HIGH at 2 GHz, a point that anticipates the memory wall in future embedded systems – notice that schedulability is universally very good for WRR with 1, 2, and 4 DRAM banks.

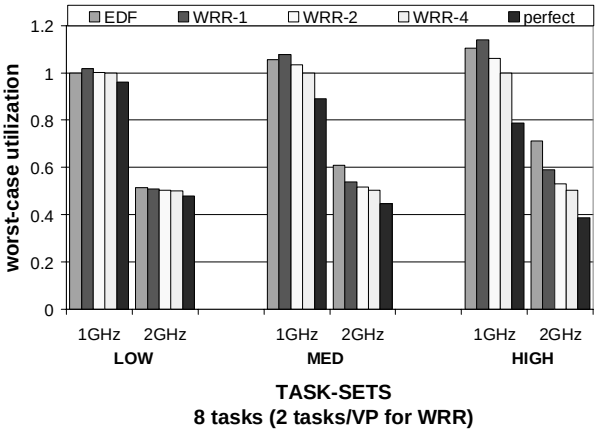

Figure 6. Varying the number of DRAM banks.

6.2 Simulation Demonstration

The graph in Figure 7 shows run-time utilization of the task-sets assuming four DRAM banks, measured by cycle-level simulation. Run-time utilization is naturally less than worst-case utilization, because it depends on actual execution times instead of worst-case execution times. All task-sets were successfully scheduled using the adapted WRR at 1 GHz, with a run-time utilization less than 1 (as predicted by schedulability analysis). On the other hand, using EDF scheduling, task-sets HIGH and MED miss their deadlines and terminate (explaining why the "EDF" bar is unavailable). Task-set LOW was successfully scheduled by EDF at 1 GHz due to the difference between WCETs and actual execution times. However, for hard-real-time system design, it is neither safe nor acceptable to use these actual execution times as inputs to worst-case schedulability tests.

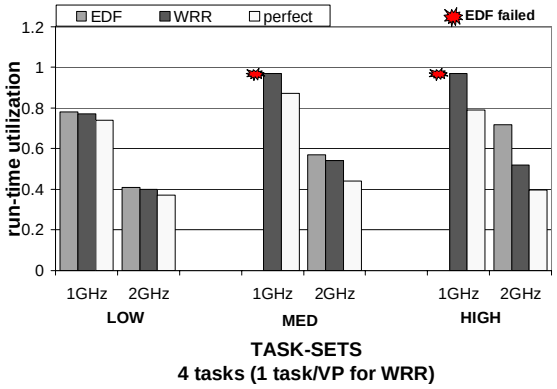

Figure 7. Run-time utilization.

7. RELATED WORK

A large body of work exists in the area of uniprocessor scheduling [16][17][21]. Liu and Layland [16] provide extensive insight into various uniprocessor scheduling algorithms such as earliest-deadline-first (EDF) and rate-monotone scheduling (RMS). Weighted-round-robin [17] is a fairly simple scheduling policy to implement, however, it perhaps receives less attention in the field of hard-real-time scheduling because of the prohibitively high context-switching overhead (WRR switches much more frequently than EDF or RMS), an aspect that improves with hardware multithreading support.

Hardware multithreading reduces the penalty of context-switching significantly, which facilitates hiding lengthy stalls due to memory accesses and even fine-grain events, such as L1 cache misses, branch mispredictions, and other ILP limiters [1][9][20][23][24][25]. However, most prior work focuses on improving average performance and bounding performance has not been a priority.

Kreuzinger et al. proposed a multithreaded processor for real-time systems and evaluated real-time scheduling policies on their substrate, e.g., fixed-priority preemptive, EDF, least-laxity first, and guaranteed-percentage [15]. They compare scheduling policies by progressively tightening periods and discarding policies that fail first, i.e., they perform dynamic testing. They do

not analytically model overlap and hence do not provide a static means for testing schedulability under all possible scenarios.

Software thread integration (STI) is a compiler technique that co-mingles instructions from multiple threads in the same binary [8]. It has not been applied to overlapping computation and memory of integrated threads. Although STI could be extended to do so, statically overlapping memory transfers of one thread with computation of another thread requires knowing the relative positioning of computation and memory, whereas our approach does not.

Jain et al. provide the first (and extensive) study of soft-real-time scheduling on SMT processors, pointing out the unique opportunities and challenges in this new setting [13]. They divide the problem into two sub-problems, co-scheduling (which tasks to run simultaneously) and resource sharing (how to share resources among co-scheduled tasks). Key factors in achieving schedulability include prioritization of high-utilization tasks and exploitation of symbiosis. They only consider soft-real-time tasks and consequently consider a task-set to be schedulable even if some fraction (5%) of deadlines are missed. Schedulability is evaluated on the basis of dynamic testing. In contrast, we consider hard-real-time tasks and provide a static schedulability test accordingly.

8. SUMMARY AND FUTURE WORK

Contemporary embedded processors are rapidly evolving in terms of clock frequency, worsening the memory wall problem even in the embedded systems domain. Moreover, classic real-time scheduling theory for uniprocessors is outdated in its inability to model irregular parallelism afforded by multithreaded processors, missing the opportunity to overlap WCETs of tasks and exceed conventional uniprocessor scheduling limits.

In this paper, we developed an analytical framework that safely and tractably bounds overlap between computation of a pipeline-resident task with memory transfers of otherwise-idle tasks. This is a key departure from prior multithreading research, which focuses on average performance instead of worst-case performance, and is therefore incompatible with hard-real-time systems. We then derived a closed-form schedulability test that only depends on the aggregate breakdown of memory and computation time in tasks' WCETs, and not the specific ordering of computation and memory transfers within and among tasks. With this new formalism, we are able to safely and tractably overlap WCETs in hard-real-time systems on multithreaded processors. From schedulability experiments with real task-sets, task-sets with perceptible memory time that are unschedulable using conventional EDF become schedulable using our adapted WRR, due to the ability to *analytically* overlap tasks' WCETs.

For future work, we plan to extend our analytical framework to include more flexible and fine-grain forms of multithreading, such as simultaneous multithreading. Further, we are exploring less conservative approaches for modeling worst-case bus and bank conflicts in the memory system.

9. ACKNOWLEDGMENTS

We thank Aravindh Anantaraman for insightful discussions, contributions to worst-case timing analysis, and comments on drafts of this paper.

This research was supported in part by NSF grants CCR-0207785, CCR-0208581, and CCR-0310860, NSF CAREER grant CCR-0092832, and generous funding and equipment donations from Intel and Ubicom.

10. REFERENCES

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, June 1990.

[2] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[3] ARM, Inc. ARM-11 Technical Reference Manual. Available from: http://www.arm.com/pdfs/DDI0211D_arm1136_r0p2_trm.pdf.

[4] D. Burger, T. Austin, and S. Bennett. The Simplescalar Tool Set, Version 2.0. Technical Report 1342, Computer Science Department, University of Wisconsin-Madison, 1997.

[5] G. Buttazzo. Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. *Kluwer Academic Publishers*, 1997.

[6] C-Lab WCET Benchmarks. Available from: http://www.c-lab.de/home/en/download.html.

[7] B. Cogswell and Z. Segall. MACS: A Predictable Architecture for Real Time Systems. In *Proceedings of the 12th IEEE Real-Time Systems Symposium*, December 1991.

[8] A. Dean and J. Shen. Techniques for Software Thread Integration in Real-Time Embedded Systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[9] R. Eickemeyer, R. Johnson, S. Kunkel, M. Squillante, and S. Liu. Evaluation of Multithreaded Uniprocessors for Commercial Application Environments. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[10] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread Level Parallelism of Desktop Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[11] T. Hand. Real-Time Systems Need Predictability. *Computer Design (RISC Supplement)*, August 1989.

[12] C. Healy, D. Whalley, and M. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the 16th Real-Time Systems Symposium*, December 1995.

[13] R. Jain, C. J. Hughes, and S. V. Adve. Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, December 2002.

[14] D. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, December 1989.

[15] J. Kreuzinger, A. Schulz, M. Pfeffer, and T. Ungerer. Real-Time Scheduling on Multithreaded Processors. In *Proceedings of the 7th International Conference on Real-Time Computer Systems and Applications*, December 2000.

[16] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment. *Journal of ACM*, vol. 20, pp. 46-61, January 1973.

[17] J. Liu. Real-Time Systems. Prentice Hall, 2000.

[18] T. Lundqvist and P. Stenstrom. An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution. *Journal of Real-Time Systems*, 17(2/3):183-208, November 1999.

[19] F. Mueller. Compiler Support for Software-Based Cache Partitioning. In *Proceedings of Programming Language Design and Implementation*, June 1995.

[20] B. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. In *Proceedings of Real Time Signal Processing IV*, 1981.

[21] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. Deadline Scheduling for Real-Time Systems. *Kluwer Academic Publishers*, 1998.

[22] S. Storino and J. Borkenhagen. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor Design. In *Proceedings of the International Symposium on High-Performance Chips*, August 1999.

[23] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[24] Ubicom, Inc. The Ubicom IP3023 Wireless Network Processor. Available from: http://www.ubicom.com/products/ip3000/ip3000.html.

[25] T. Ungerer, B. Robic, and J. Silc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, Vol. 35, No. 1, March 2003.

[26] A. Wolfe. Software-Based Cache Partitioning for Real-Time Applications. In *Proceedings of the 3rd International Workshop on Responsive Computer Systems*, September 1993.