On the Near-Optimality of List Scheduling Heuristics for Local and Global Instruction Scheduling

by

John Michael Chase

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

Abstract

Modern architectures allow multiple instructions to be issued at once and have other complex features. To account for this, compilers perform instruction scheduling after generating the output code. The instruction scheduling problem is to find an optimal schedule given the limitations and capabilities of the architecture. While this can be done optimally, a greedy algorithm known as list scheduling is used in practice in most production compilers.

List scheduling is generally regarded as being near-optimal in practice, provided a good choice of heuristic is used. However, previous work comparing a list scheduler against an optimal scheduler either makes the assumption that an idealized architectural model is being used or uses too few test cases to strongly prove or disprove the assumed near-optimality of list scheduling. It remains an open question whether or not list scheduling performs well when scheduling for a realistic architectural model.

Using constraint programming, we developed an efficient optimal scheduler capable of scheduling even very large blocks within a popular benchmark suite in a reasonable amount of time. I improved the architectural model and optimal scheduler by allowing for an issue width not equal to the number of functional units, instructions that monopolize the processor for one cycle, and non-fully pipelined instructions. I then evaluated the performance of list scheduling for this more realistic architectural model.

I found that when scheduling for basic blocks when using a realistic architectural model, only 6% or less of schedules produced by a list scheduler are non-optimal, but when scheduling for superblocks, at least 40% of schedules produced by a list scheduler are non-optimal. Furthermore, when the list scheduler and optimal scheduler differed, the optimal scheduler was able to improve schedule cost by at least 5% on average, realizing maximum improvements of 82%. This suggests that list scheduling is only a viable solution in practice when scheduling basic blocks. When scheduling superblocks, the advantage of using a list scheduler is its speed, not the quality of schedules produced, and other alternatives to list scheduling should be considered.

Acknowledgments

I would like to thank my supervisor, Peter van Beek, for his guidance, mentoring, and support throughout my work as an undergraduate and graduate student with him. Thanks to Tyrel Russell and Abid Malik for their many contributions in the discussions we had as a research group. I am also appreciative of Farhad Mavadat and Ondrej Lhotak for serving on my committee.

I would also like to thank my family and friends, and especially my wife, Jen, for encouraging and supporting me in my research and also helping me to balance between school and other aspects of my life.

This work was made possible by IBM Corp. and by the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca).

Dedication

This work is dedicated to Jen, for her constant support and love and for all that she did in order to allow me to pursue this degree.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern architectures are capable of issuing and executing multiple instructions at once, as well as having many other interesting properties. The code generation phase of a compiler produces a straight-line sequence of code which must be scheduled in order to execute multiple instructions at once and take advantage of other architectural features. This process, known as instruction scheduling, is performed in practice by a non-optimal, greedy algorithm known as the list scheduling algorithm.

List scheduling is widely assumed to be nearly optimal with an appropriate choice of heuristic with which to rank instructions that are ready to be issued. However, any previous work that compares the list scheduling algorithm against an optimal scheduler either uses a small set of test data, producing inconclusive results, or makes many simplifying assumptions about the architectural model upon which the scheduled code will be executed. It is an open question whether or not list scheduling is nearly optimal when scheduling for a realistic architectural model. Especially for embedded processors with limited computing power, it is essential that the compiled code be as efficient as possible. If the list scheduling algorithm does not provide good enough schedules, it may be necessary to consider other scheduling algorithms.

In this thesis, I take an optimal scheduler that initially assumes the same simple architectural model as list scheduling and improve the model, making it more realistic. I then compare the schedules produced by the list scheduling algorithm against those produced by the optimal scheduler. By using a realistic architectural model and evaluating with a sufficiently large set of test data, I provide conclusive evidence that speaks to the near-optimality of list scheduling. When scheduling for basic blocks, the list scheduling algorithm produces optimal schedules at least 94%

of the time for the target architectures on which it was evaluated. When scheduling for superblocks, list scheduling produces optimal schedules between 47% and 60% of the time. The difference between the list scheduler and optimal scheduler is also significant: improved blocks cost between 5% and 8% less on average, with a maximum improvement of 82%.

These results suggest that list scheduling performs sufficiently well on more complex architectures for local instruction scheduling only, and that global instruction scheduling algorithms must be given further consideration in order to produce better overall schedules.

1.1 Contributions

The most significant contribution of this thesis is that list scheduling is proven to be far from optimal when scheduling superblocks when the target architectural model is complex, although list scheduling basic blocks produces near-optimal results for a complex architectural model. When scheduling basic blocks, the list scheduling algorithm was optimal over 98% of the time for an idealized architectural model and over 94% of the time for a realistic architectural model. However, the list scheduler performs poorly when scheduling superblocks for a realistic architectural model, with between 39.9% and 52.3% of schedules produced being non-optimal and an average improvement of 5.3%-8.1%. These results are important because superblocks are frequently executed sequences of instructions, and any speed improvement on these will almost certainly result in a speed improvement for the whole program.

This thesis contains some other relevant contributions. When comparing a heuristic given by Shieh and Papachristou [35] against the more common critical path heuristic, Shieh and Papachristou's heuristic generally performed better. As both heuristics use critical path as the primary feature, this suggests that the choice of secondary features is important to instruction scheduling heuristics, which provides motivation for a re-evaluation of existing heuristics. There is also little difference in terms of register usage between schedules produced by a list scheduler and our optimal scheduler. This is significant as it shows that not only does list scheduling produce near-optimal schedules, it also produces schedules with similar register requirements as an optimal scheduler and there is thus little purpose for using an optimal scheduler.

1.2 Overview

This thesis is divided into six chapters. In Chapter 2, I formalize the instruction scheduling problem and the list scheduling algorithm, present directed acyclic graphs for instruction scheduling, discuss heuristics and features for list scheduling, and provide an introduction to constraint programming. Chapter 3 surveys work done evaluating the optimality of the list scheduling algorithm. In Chapter 4, I examine instruction scheduling for basic blocks, presenting the initial architectural model as well as several improvements to it. I compare the list scheduler to an optimal scheduler for several architectures using both the initial and improved architectural models in order to evaluate the optimality of the list scheduling algorithm. In Chapter 5, I repeat the work done in Chapter 4 but I instead consider instruction scheduling for superblocks. I conclude in Chapter 6, summarizing the results found in this thesis and discussing possibilities for future work.

Chapter 2

Background

The code generation phase in a compiler typically produces a straight-line sequence of code. The code generation phase is often followed by an instruction scheduling phase which rearranges the code to achieve better performance on modern architectures.

Modern architectures have multiple *functional units*, processing units dedicated to a specific type of instruction, such as integer or branch instructions. In addition, some architectures allow multiple instructions to be issued in a single cycle. The maximum number of instructions that can be issued in one cycle is known as the processor's *issue width*. The issue width for a particular architecture must be less than or equal to the number of functional units.

Each instruction on a particular architecture has an execution time and a latency. *Execution time* is the number of cycles for which an instruction locks up a functional unit to the exclusion of all other instructions. The *latency* of an instruction is the number of cycles needed after an instruction is issued before its result is available to other instructions. An instruction's latency is always greater than or equal to its execution time. An instruction can also be *fully pipelined*: if its execution time is 1, the functional unit is only occupied by the instruction for the cycle in which the instruction was issued. Instructions with execution times greater than one are said to be *not fully pipelined*.

Modern architectures are *pipelined*: an instruction with a latency greater than one need not tie up the functional unit on which it is executing until the result is ready [18]. Architectures use pipelining to overlap the execution of instructions. If an instruction with an execution time of 2 and a latency of 5 is issued in cycle 1, another instruction can be issued on the same functional unit as early as cycle 3, even though the result from the first instruction is not available for use until cycle 6.

As an example of an architecture with all these properties, consider the PowerPC 604 [19]. It has six functional units: a branch unit, two integer units for simple instructions, an integer unit for more complex instructions, a floating point unit, and a load/store unit for data transfer to and from memory. The PowerPC 604 has an issue width of 4, so not every functional unit will begin executing a new instruction every cycle. Most instructions are fully pipelined, and thus the PowerPC 604 can often dispatch a new instruction for execution each cycle on the same functional unit. However, some instructions are not fully pipelined and monopolize a functional unit for the entire duration of their execution. One such example for the PowerPC architecture is *divw*, one of many integer division instructions. It has an execution time of 19 cycles on the PowerPC 604 and has a latency of 20 cycles. There is thus one cycle occurring after the issue of a divw instruction for which the instruction is not executing but its result is unusable.

2.1 Scheduling Units

The most common straight-line unit of code is the *basic block*, a code sequence having a single entry point and a single exit point [29], where the only branching that occurs is a branch from the exit instruction to another basic block, or to the same basic block in order to create a loop. Not all compilers produce basic blocks having a single entry point, but this can be remedied by inserting a "dummy" instruction with zero execution time, and forcing all existing entry point instructions to follow the dummy instruction. A similar method can be used to ensure that basic blocks all have a single exit point.

The other common scheduling unit used in compilers is the *superblock* [11, 20]. A superblock consists of a series of basic blocks $B_0, B_1, \ldots, B_n$ such that:

- Each basic block either branches out of the superblock or branches to the next basic block in sequence (i.e. for $k = 0, \ldots, n-1$, B_k branches out of the superblock or branches to B_{k+1})
- There are no branches from any block B_i to any other block B_j, for $j \neq 0$, other than the branch from B_i to B_{i+1}. (i.e. there are no cycles in the superblock that do not pass through block B_0)

- There are no branches into any basic block within the superblock except for branches to B_0. These disallowed branches are known as *side entrances.*

Branch instructions in a superblock that both transfer control from the superblock and occur before the final instruction are known as *side exits*, and the final instruction in the superblock is known as the *final exit.* Superblock scheduling often makes use of profiling information for the program being compiled. Each side exit in a superblock can be assigned a probability that the side exit is taken. If the side exit is not taken, control proceeds to the following basic block. The final exit is always taken if it is reached. Branch probabilities are given in the form $branch(i) = (P(i), 1 - P(i))$, for $0 \leq P(i) \leq 1$, where $P(i)$ is the percentage of times the branch was taken given that the branch condition is evaluated. In other words, if a side exit has branch probability $(0.3, 0.7)$, then the branch is taken 30% of the time the branch instruction is executed and not taken 70% of the time.

The formula for evaluating schedule length for superblocks given in Section 2.3 requires a cost coefficient for each exit i. This coefficient is the probability that branch i is taken given that branches $0 \leq i-1$ were not taken. For example, suppose a superblock has two side exits. The first has branch probability (0.2, 0.8), and the second has branch probability (0.6, 0.4). The cost coefficient for the first side exit is 0.2, since there is a 20% chance the first branch is taken. The cost coefficient for the second side exit is $0.8 \times 0.6 = 0.48$, since the first branch is not taken 80% of the time and the second branch is taken 60% of the time. The final exit is always taken if it is reached, and so its cost coefficient is $1 - 0.2 - 0.48 = 0.32$: a side exit is taken 68% of the time, and so the final exit must be taken the remaining 32% of the time.

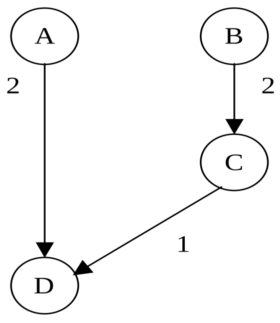
Throughout this thesis, I refer to the probability that a branch is taken given that the branch instruction is executed as the *branch probability*, and the probability that a branch is taken given that all previous branches in the superblock were not taken as the *exit probability.* I use $branch(i)$ and $exit(i)$ to denote the branch and exit probabilities respectively for instruction i.

While basic blocks are considered to be local scheduling units and superblocks are labelled as "global" scheduling units, they are not global in the sense that they incorporate an entire program. Superblocks are formed by first finding a *trace* [12]: a region of instructions identical to superblocks except that side entrances are permitted. To eliminate side entrances from the trace, the code following a side entrance is duplicated, known as *tail duplication* [20]. The side entrances branch to the old copy of the code, and the trace branches to the new copy of the code. When all side entrances are removed in this manner, the trace is now a superblock. Superblocks are preferable to traces in that traces require a significantly more complex compiler implementation in order to deal with side entrances, while superblocks do not [11].

For simplification, this thesis refers to a *block* when the context allows for both basic blocks and superblocks without distinction.

2.2 Instruction DAGs

A common conceptual view of an instruction scheduling problem is a directed acyclic graph, or DAG (see [29]), also known as a Program Dependence Graph or Data Dependence Graph in the compiler literature. Figure 2.1 shows a sample DAG from a basic block selected from our testing data, and Figure 2.2 shows a DAG for a superblock, also selected from testing data. In an instruction DAG, each node corresponds to an instruction from the original straight-line basic block or superblock. Nodes are labeled alphabetically, beginning with A. Nodes are also assigned a sequence number 1 through n, with node i being the i^{th} instruction in the order given in the original block. If instruction i must be completed before instruction j in the original block, an edge is added between nodes i and j. This is known as a *precedence constraint*: i has precedence over j, and must be scheduled first in any correct schedule.



(a)

1. (A) or. gr1, gr2, gr2
2. (B) addis gr3, 0, 15
3. NOP
4. (C) stw gr4, 424(gr3)
5. (D) bc 2, cr0, 132

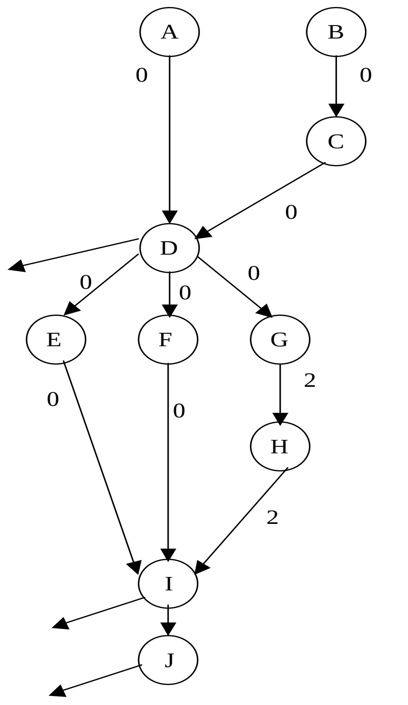
(b)

1. (B) addis gr3, 0, 15
2. (A) or. gr1, gr2, gr2
3. (C) stw gr4, 424(gr3)
4. (D) bc 2, cr0, 132

(c)

Figure 2.1: (a) DAG for a basic block from the SPEC 2000 compiler benchmark; (b) One possible schedule for a single-issue processor, where NOP denotes a cycle in which no instructions are scheduled; (c) An optimal schedule.

Edges in the DAG are also assigned a weight. If an instruction j must execute at least $l(i, j)$ cycles after instruction i begins execution, edge (i, j) is assigned weight



(a)

1. (D) 0.5
2. (I) 0.8
3. (J) 1.0

(b)

1. (D) 0.5
2. (I) 0.4
3. (J) 0.1

(c)

Figure 2.2: (a) DAG for a superblock from the SPEC 2000 compiler benchmark; (b) Branch probabilities for side exits and the final exit; (c) Exit probabilities for side exits and the final exit.

$l(i, j)$. This is known as a *latency constraint*, and $l(i, j)$ is said to be the *latency* [29] of instruction i. If $l(i, j) = 1$, instruction j can begin execution in the cycle immediately following instruction i. Other instructions can be issued between the cycles in which instructions i and j begin, provided all constraints are satisfied

On the PowerPC 604 architecture, most instructions have a latency of 1 or 2, and the maximum latency of any instruction is 32. Only floating point instructions have a latency over 10; the majority of instructions produce their results in very few cycles. However, many edges with long-running instructions as the source node, including the 32-cycle *fdiv* instruction, have low latencies. This phenomenon is explained by Smotherman et al. [37]. Suppose an instruction i has an execution time of 3 cycles and a latency of 4 cycles. Any instruction j needing to read the result of i must wait at least 4 cycles after i begins execution to assure that i has written its result. This is known as a *read-after-write* or *RAW* dependency, and edge (i, j) would have weight 4. Suppose instead that j writes to a register r that

i must read. If i takes only one cycle to read from r, j may be issued in the cycle following i, and so edge (i, j) would have weight 1: this is known as a *write-after-read* or *WAR* dependency. Edges may not reflect the true latency of an instruction for other reasons as well, and it is not the case that every edge in a DAG has the latency of the first node as its weight.

There is also one other type of scheduling constraint that is not explicitly captured in a DAG, because they are general constraints for the target architecture, not specific constraints for a particular block. *Resource constraints* [29] are any constraints caused by the number and type of resources on a processor. For example, if a processor has two integer units, at most two integer instructions can begin execution each cycle. Resource constraints cannot be reflected in a DAG since a family of architectures may have the same latencies for each instruction but have different resources. Thus, if a DAG is scheduled on several different architectures, schedules of different lengths may be produced.

2.3 The Instruction Scheduling Problem

To make the most of the advanced features of modern architectures, compilers perform *instruction scheduling* as an optimization after code generation. The goal of instruction scheduling is to find a minimum-cost schedule for a straight-line sequence of code, subject to several types of constraints [11, 29]. I present two versions of the instruction scheduling problem: the local instruction scheduling problem for basic blocks and the global instruction scheduling problem for superblocks. As mentioned in Section 2.1, the latter is not truly global. As the two differ only slightly, I first give a general version of the instruction scheduling problem adapted from [38], and refine it to account for the differences between the two scheduling problems. The definition of the instruction scheduling problem makes the assumptions that all instructions are fully pipelined and that either the machine has an infinite number of registers or that register allocation has taken place before instruction scheduling.

Definition 1 (Instruction Scheduling Problem) Consider a DAG $G = (V, E)$ representing a block, where each edge (i, j) has weight $l(i, j)$. The target architecture has a global issue width W and a set of functional units U. There is a set T of types of functional units, and each unit is of some type $t \in T$. There are $f(t)$ functional units of type $t \in T$, and each instruction i is of type $u(i)$, where $u(i) \in T$. Let y_{ik} be a binary variable that takes on the value 1 if and only if instruction i is scheduled in cycle k. A feasible schedule S specifies an issue time

$S(i)$ for all instructions i. $S(i)$ and y_{ik} are related*:* $\forall i \forall k, y_{ik} = 1$ *iff* $S(i) = k$.S must also satisfy the following constraints*:*

- $\forall k, W \geq \sum_{i=1}^{n} y_{ik}$ *(global issue width constraint)*
- $\forall k, t \in T, f(t) \geq \sum_{u(i)=t} y_{ik}, 1 \leq i \leq n$ *(functional unit constraints)*
- $\forall (i,j) \in E, S(j) \geq S(i) + l(i,j)$ *(latency constraints)*

The Instruction Scheduling Problem is to find a schedule S for which a cost function $cost(S)$ is minimized.

There are several cost measures for the instruction scheduling problem. The actual execution time on a physical architecture is one such measure, although it is hard to evaluate while scheduling is performed. For basic blocks, the common metric is schedule length. The sooner all instructions in a DAG finish executing, the shorter the schedule will be. Schedule length can be thought of as the latest cycle in which an instruction is issued: that is, $cost(S) = \max_{i \in V}\{S(i)\}$. Figure 2.1 (b) shows a valid schedule for the DAG in Figure 2.1 (a), but schedule (b) is clearly suboptimal as a better schedule is presented in Figure 2.1 (c).

Definition 2 (Local Instruction Scheduling Problem) The Local Instruction Scheduling Problem is to solve the Instruction Scheduling Problem where the DAG G represents a basic block, minimizing cost function $cost(S) = \max_{i \in V}\{S(i)\}$.

When scheduling superblocks, the measure of evaluation is the expected number of cycles executed within the superblock before a branch is taken or the final exit is reached.

Definition 3 (Superblock Schedule Cost) Let $exit(i)$ be the exit probability for node i in the DAG, where $0 \leq exit(i) \leq 1$ *and* $exit(i) = 0$ when node i is not a side exit or the final exit. For a schedule $S, cost(S) = \sum_{i=1}^{n} exit(i)S(i)$.

Definition 4 (Global Instruction Scheduling Problem) The Global Instruction Scheduling Problem is to solve the Instruction Scheduling Problem where the DAG G represents a superblock, minimizing cost function $cost(S) = \sum_{i=1}^{n} exit(i)S(i)$.

2.4 List Scheduling

Finding an optimal solution to both the local and global instruction scheduling problems is NP-complete [17]. The emphasis of scheduling research has been on approximation algorithms. Hennessy and Gross [17] made an early attempt at developing a polynomial algorithm for instruction scheduling. Their algorithm had a worst-case runtime of $O(n^4)$. Gibbons and Muchnick [14] were able to refine the algorithm to provide a worst-case runtime of $O(n^2)$, although the quality of schedules produced is not necessarily as good. This refined algorithm, known as the *list scheduling* algorithm, has become the most popular instruction scheduling algorithm, and is used almost exclusively in production compilers.

The list scheduling algorithm is so-called because of its use of a *ready list*. The algorithm iterates through machine cycles sequentially, and at each cycle it populates the ready list with the set of all *candidate* instructions which could begin execution in the current cycle. It then selects the best instructions from the ready list to begin execution in the current cycle, subject to resource constraints [29]. The algorithm also makes use of an *execution list*: whenever an instruction is issued, it is placed on the execution list, a list of all instructions currently being executed. When an instruction i finishes executing, any successor j of i becomes a candidate instruction as long as all other predecessors of j have also finished executing. The execution list is used to easily identify instructions that finish executing, so it can quickly be determined if j may be added to the ready list. Selection is made according to a heuristic independent of the core list scheduling algorithm. In Algorithm 1, I present a formal representation of the list scheduling algorithm, adapted from the presentation in [34].

The method **selectBestInstruction** encapsulates the particular heuristic used in an implementation of list scheduling. For each instruction in the ready list in descending order of priority according to the heuristic, it determines whether or not the instruction can be executed in the current cycle. In other words, it checks whether there is an available functional unit of the instruction's type and whether or not the number of instructions scheduled to begin in the current cycle is less than or equal to the global issue width. If there is an available functional unit and the global issue width will not be exceeded by issuing the current instruction, that instruction is returned. If there is no instruction on the ready list that can begin execution in the current cycle, **selectBestInstruction** returns null.

It is worth pointing out that the list scheduling algorithm pays no attention to any other architectural features or hazards besides available functional units and issue width. In particular, instruction cache and data cache misses are ignored,

Algorithm 1: List Scheduling

```
input  : A DAG $G = (V, E)$, global issue width $W$, number of functional
         units $f(t)$
output: A valid schedule satisfying the precedence constraints of $G$ and the
         architectural constraints of $W$ and $f(t)$
$cycle = 0$;
$ready\text{-}list$ = all source nodes in $G$;
$execution\text{-}list$ = empty;
while ( $ready\text{-}list$ or $execution\text{-}list$ are not empty ) do
    $op_i$ = selectBestInstruction( $ready\text{-}list$ );
    while ( $op_i$ is not null ) do
        remove $op_i$ from $ready\text{-}list$ and add to $execution\text{-}list$;
        for all instructions $op_j$ such that $(op_i, op_j) \in E$ do
            Add $op_j$ to $ready\text{-}list$ if $op_j$ is ready to be executed;
        $op_i$ = selectBestInstruction( $ready\text{-}list$ );
    $cycle = cycle + 1$;
    for ( $op_i$ = all nodes in $execution\text{-}list$ ) do
        if ( $op_i$ finishes in cycle $cycle$ ) then
            remove $op_i$ from $execution\text{-}list$;
            for all instructions $op_j$ such that $(op_i, op_j) \in E$ do
                Add $op_j$ to $ready\text{-}list$ if $op_j$ is ready to be executed;
```

and the algorithm assumes all instructions are fully pipelined. These assumptions do not affect the correctness of the schedules produced, as architectures are able to introduce appropriate stalls when necessary. It is possible that better schedules might be obtained if these assumptions were accounted for in the algorithm. However, the list scheduling algorithm employed in many, if not all, production compilers is essentially the same as Algorithm 1. The main difference is the choice of heuristic used in **selectBestInstruction**.

Algorithm 1 has a worst-case runtime of $O(n^2)$, demonstrated by the analysis presented in [34]. Each of the n instructions in the DAG is added once to the ready list, and is chosen once by **selectBestInstruction**. Assuming the ready list is implemented using a heap-based priority queue and that only static features are used (see Section 2.4.1), **selectBestInstruction** has a runtime of $O(\log n)$, and the code preceding the second *for* loop runs in $O(n \log n)$ time. Within the second **for** loop, each instruction is removed once from the execution list, and for each instruction, all outgoing edges must be checked. The DAG contains E edges, and so checking all outgoing edges can be done in $O(E)$ time. The overall runtime of the algorithm is thus $O(E + n \log n)$, which can be $O(n^2)$ in the worst case (as $|E|$ can be $O(n^2)$). In practice, however, DAGs are usually sparse, and the runtime is closer to $O(n + n \log n)$. In our work, we use an iterative scan of the ready list, instead of a priority queue. Scanning the ready list thus has a worst case runtime of $O(n^2)$ in our implementation, but since in practice the ready list is usually short, we achieve acceptable performance, especially when the list scheduler is compared to our optimal scheduler, which has a much higher computational time than list scheduling for any basic block.

2.4.1 Scheduling Heuristics and Features

When the list scheduler chooses an instruction to be scheduled in the current cycle, it uses a *heuristic* to choose the best instruction on the ready list if there is more than one instruction that can be scheduled [29]. Each instruction has a set of *features*, where a feature is a significant property of the instruction. Features may be *static*, meaning that their value never changes over the course of scheduling, or they may be *dynamic*, indicating that their value may change during scheduling. A list scheduling heuristic is a function that takes as input a pair of instructions, and based on the features of those instructions, gives a preference of one instruction over the other. It need not be the case that all possible features for instructions are used in a heuristic; if two instructions agree on every feature in the heuristic, one is chosen arbitrarily.

Smotherman et al. [37] provide a survey of common scheduling heuristics used for local instruction scheduling. They also describe a large number of features that can be used for both local and global instruction scheduling. I describe the features used in the heuristics we compare against here; the reader is referred to [37] for coverage of other features.

Critical Path Distance to Sink: The critical path distance between two nodes in a DAG is defined as the maximum length path between the two nodes, where path length is the sum of latencies encountered along the path. Critical path distance to sink refers to the distance between any node and the sink in the DAG. This feature is what is generally meant by "critical path," and is labelled as such throughout the remainder of this thesis. For example, the critical path distance from node B to node D in the DAG in Figure 2.1 is 3.

Dependence Height: The dependence height of a node in a DAG is the number of nodes on the longest path from the node to the sink [13].

Earliest Starting Time: A static estimate of the earliest cycle in which an instruction can begin execution. The root node has an EST of 1, the first cycle for scheduling. Any other node j has an EST of $\max\{l(i,j) + EST(i)\}$ for any parent i of j.

Instruction Type: There is no clear notion of a "best" or "worst" instruction type. In our critical path heuristic (refer to Table 2.1), we favour floating point instructions above all other instructions and treat remaining instructions equally with respect to their instruction type, as done in [3].

Maximum Latency: This feature is simply the maximum latency of an edge from a DAG node to any other DAG node. Maximum latency can also be thought of as the longest possible time that any other node in the DAG will have to wait for a result from the current node once the current node begins execution.

Number of Successors: The total number of nodes reachable by following a single edge from the current node.

Local Instruction Scheduling Heuristics

In my work on basic block scheduling, I compare against two heuristics. The first is a critical path heuristic based on the one used in IBM's TOBEY Compiler [3]. The second heuristic is one presented by Shieh and Papachristou [35]. Critical Path Distance is widely accepted as the most significant feature among standard instruction features for heuristic-based scheduling in the literature (for example, [3] and [29]), but the choice of less-significant features for tie-breaking varies widely. Shieh and Papachristou's heuristic relies on secondary features that differ significantly from other common critical path distance-based heuristics, and I included it to compare the performance of critical path-based heuristics using different secondary features. Table 2.1 shows the ranking of the features used in these heuristics.

Global Instruction Scheduling Heuristics

I compare two list scheduling heuristics against an optimal scheduler, as I did for local instruction scheduling. One heuristic is the critical path heuristic presented previously, and the other is Dependence Height and Speculative Yield (DHASY) [5].

DHASY is based on a combination of the Dependence Height and Speculative Yield features suggested first by Fisher [13]. Dependence Height is defined in Section 2.4.1, and Fisher's speculative yield function is equivalent for superblocks to

Table 2.1: Local instruction scheduling heuristics, showing rank of features for each heuristic. For a pair of instructions, the instruction with the better value for the feature ranked number 1 is selected. If the two instructions have the same value for that feature, the instruction with the better value for the rank 2 feature is selected, and so on.

	Critical	Shieh and
Feature	Path	Papachristou
Critical Path Distance	1	1
Earliest Starting Time	3	
Instruction Type	2	
Maximum Latency		2
Number of Successors		3

the exit probability described in Section 2.1 [8]. The DHASY value is calculated for each instruction i, and the instruction with the highest value is selected for scheduling. Deitrich and Hwu [8] formalize DHASY mathematically as follows:

$$dhasy(i) = \sum_{b \in B_i} (exit(b)(cp(1,n) + 1 - (cp(1,b) - cp(i,b)))$$

where B_i is the set of all branches that are descendants of i, and $cp(i,j)$ is the critical path distance from i to j.

Some other heuristics that were considered were Balance Scheduling [9], G* [6], Speculative Hedge [8], and Successive Retirement [6]. DHASY performed better than G* and Successive Retirement both theoretically [9] and in our experience [32], although Shobaki and Wilken [36] have experimental results suggesting Successive Retirement outperforms DHASY. Balance Scheduling is reported to perform better than DHASY [9], as is Speculative Hedge [8, 9]. However, Russell [32] found that when a larger benchmark suite was used for comparison, in this case the complete SPEC 2000 benchmark suite, DHASY ended up being a better heuristic than Speculative Hedge. While Balance Scheduling still performed better, his implementation of Balance Scheduling ran significantly slower than his implementation of DHASY and was much more complicated. I selected DHASY as the second heuristic for comparison because of its superior performance, both in terms of execution time and schedule cost, when compared to other heuristics which account for side exits and their impact on schedule cost.

Critical Path was chosen as it is a standard heuristic for superblocks, even though it does not explicitly consider the cost function and may select other instructions when a side exit is on the ready list. As the critical path heuristic predates the superblock, newer heuristics should provide better performance than critical path. Selecting critical path as a heuristic for comparison thus provides a standard with which to evaluate the quality of other heuristics.

2.5 Register Pressure

Instruction scheduling cannot be done in isolation. Most instructions use one or more registers, either as source registers containing values used in a computation or as destination registers which store the results of a computation. The code generation phase of most compilers produces an instruction sequence using an infinite number of virtual registers. As there are a limited number of physical registers on an architecture, a compiler must perform *register allocation* to assign each virtual register to a physical register.

Register allocation and instruction scheduling are subject to a phase-ordering problem. If instruction scheduling is performed first, the resulting schedule may require more physical registers than those available. This in turn forces some *spilling* to occur: the values in one or more registers must be copied, or spilled, to memory so the registers are available for use, possibly lengthening the schedule [29]. Alternately, if register allocation is performed first, the limitations on available registers may adversely impact the quality of the schedule produced.

The *live range* of a register in a given schedule is the set of cycles for which the register is active and contains a single value. In other words, a register is live from the cycle in which it is assigned a value and remains live until the last cycle in which that value is used. This makes the assumption that values are not live across block boundaries, an assumption that must be made for my purposes as I do not have data indicating which registers are live across block boundaries. The live range of a register need not be a contiguous set. This situation arises when a variable is assigned a value more than once in a block. The register does not remain live between the last use corresponding to the first value and the assignment of the second value, as the first value is no longer needed after its last use. As an example, the live range $\{1, 2, 3, 7, 8, 9, 10\}$ indicates that the corresponding register is assigned a value in cycle 1 which is last used in cycle 3, and that the same register is reused to store a different value in cycle 7, which is last used in cycle 10.

In Figure 2.1, register gr3 is live for 3 cycles in both schedules given. In the non-optimal schedule, the live range of gr3 is $\{2, 3, 4\}$; in the optimal schedule, its live range is $\{1, 2, 3\}$.

The *register pressure* of a schedule is the maximum number of registers that are simultaneously live. In other words, if k is the value that occurs most frequently in the live ranges of all registers used in the schedule, the register pressure is the number of registers that contain k in their live range. The register pressure of a schedule is also the minimum number of physical registers that must be available for the schedule in order to avoid spilling.

I do not look directly at register allocation in this thesis, nor have I modelled register usage or incorporated any register-related information into the list scheduler or optimal scheduler, as that is beyond the scope of this thesis. However, this thesis does examine the register pressure of the schedules produced so the list scheduler can be evaluated compared to our optimal scheduler both for schedule cost and register pressure.

2.6 Constraint Programming

In our work, we also model an instruction DAG as a constraint satisfaction problem (CSP). The process of solving a CSP is known as *constraint programming*; for an appropriate model of a DAG, this allows us to find a valid schedule corresponding to the DAG.

A CSP consists of a set of n variables $\{X_1, X_2, \ldots X_n\}$, and a set of m constraints $\{C_1, C_2 \ldots C_m\}$ [31]. For each variable X_i, there is a set of values $dom(X_i)$ that can be assigned to X_i. The object of constraint programming is to find a solution to the CSP, where a solution is a valid assignment for each variable. That is, a solution is an assignment $\{X_1 = v_1, X_2 = v_2, \ldots X_n = v_n\}$ where $v_i \in dom(X_i)$ for all i. Some constraint problems also have an *objective function*: a function of some or all variables that must be minimized or maximized by the assignment.

A constraint is an expression limiting the values of one or more variables. For a constraint C, I define $vars(C)$ to be the set of variables used in that constraint. For a constraint C_j and a value $k \in dom(X_i)$, where $X_i \in vars(C_j)$, there is a *support* for k in C_j if there exists a value for all other variables $X \in vars(C_j) - \{X_i\}$ such that C_j is satisfied, where the value for each other variable X, the value is selected from $dom(X)$.

Unary constraints, such as $X_2 < 5$, affect only one variable, and binary constraints, such as $X_6 \geq X_5 + 9$, constrain two variables. Another common type of constraint is a *global constraint* [30]: a constraint on many or all variables. The best-known global constraint is the all different constraint, which specifies that no pair of variables in a set can share the same value (see [1]).

A special form of global constraint is a *global cardinality constraint* [30]. Global cardinality constraints take the form $gcc(X_1, \ldots, X_m, c_{v_1}, \ldots, c_{v_{m'}})$. This definition involves m variables $X_1, \ldots, X_m$ and m' count variables $c_{v_1}, \ldots, c_{v'_m}$. The set of variables is any subset of variables from the CSP, and each count variable c_{v_i} has an associated value v_i. The global cardinality constraint is a concise way to state that for each value v_i, there must be exactly c_{v_i} variables from the set $X_1, \ldots, X_m$ that are assigned value v_i for some value in $dom(c_{v_i})$. The alldifferent constraint mentioned above is a special case of the global cardinality constraint where each count variable has domain $dom(c_{v_i}) = \{0, 1\}$.

CSPs are generally solved using a backtracking search algorithm. At each branching point, a variable is selected for branching, and the variable is assigned a value from its domain. Before choosing another variable, constraints are *propagated*: knowing the value of one variable may make it possible to exactly determine the value of other variables or to reduce their domains [31]. For example, if X_1 has domain $\{1, 2, 3, 4, 5\}$, and X_2 has domain $\{1, 2, 3\}$, and X_1 is assigned the value 3, we can use the constraint $X_1 > X_2$ to reduce the domain of X_2 to $\{1, 2\}$. If, at any point, a variable has an empty domain, constraint propagation would find that the current partial solution is *inconsistent*. In this case, the algorithm backtracks, trying to assign a different value to the last assigned variable.

Chapter 3

Related Work

This chapter examines two areas of related work. List scheduling is now widely accepted as being both efficient and near-optimal, which was caused in part by theoretical work placing bounds on the quality of schedule produced. A more current technique is to evaluate the optimality of list scheduling by comparing schedules against those produced by an optimal scheduler. Although no single paper presents definitive results that rigorously justify the popularity of list scheduling in compilers, this chapter surveys both theoretical and empirical results that lend evidence to this wide-spread opinion.

3.1 Theoretical Results

Under certain conditions, list scheduling is known to be within a factor of two of optimality. R. L. Graham [15] describes an instance of a multi-processing system with n identical processors, $P_1, \ldots, P_n$, a set of tasks $T = \{T_1, \ldots, T_r\}$, and an ordered priority list $L = (T_{i_1}, \ldots, T_{i_r})$ containing each task in T. Each task T_j has a processing time of $\mu(T_j)$, and each processor must fully complete a task before beginning another task; Graham's system allows for no preemption. There is also a set of precedence constraints $\prec$ such that if $T_i \prec T_j$, task T_i must be processed before task T_j. Cast as an instruction scheduling problem, this corresponds to a processor with n identical functional units without any pipelining, and a static ready list. Under these conditions, Graham proves an upper bound of $2 - 1/n$ on the ratio of schedule length to optimal schedule length.

Graham then considers the implications of a dynamic priority list, and proves upper bounds using two heuristics. In the first heuristic, whenever a processor must choose a task, it chooses the task that heads the maximum-length chain of unscheduled tasks, namely critical path distance. The second heuristic selects the task T_i for which $\mu(T_i) + \sum_{T_i \prec T_j} \mu(T_j)$ is maximal. In both cases, upper bounds are proven to be $2 - 2/(n+1)$.

Bernstein and Gertner [2] present a polynomial algorithm that exactly solves the instruction scheduling problem for the special case where the maximum latency, m, of any instruction is 2. Their algorithm labels vertices from leaves to root using a heuristic related to maximum successors, and then they invoke list scheduling to produce a schedule using the labels in descending order as a heuristic. Their result also acts as an approximation algorithm when $m > 2$, with the schedule produced having length within $2 - 2/m$ times that of an optimal schedule.

3.2 Empirical Results

Many previous papers have empirically examined the quality of schedules produced by list schedulers. The most significant and most common question asked is how list schedulers compare in practice to the optimal schedule for real problems. Thus, a common metric used when examining optimal schedulers is to compare against list scheduling, giving concrete evidence of the quality of list scheduling. This chapter examines two classes of experiments: those that assume a simplistic architectural model and those that assume a more complex architectural model or that model an actual architecture. Each paper mentioned in this section refers to basic block scheduling only unless otherwise noted.

3.2.1 Idealized Architectures

Many experiments involving optimal instruction schedulers adopt a simplified architectural model. One rationale for this decision is that if a cumbersome but optimal technique it not efficient enough for simpler models, there is little point adapting the model for more complicated architectures. An idealized architecture contains assumptions that differ from physical architectures. The most common assumptions are that the architecture is fully pipelined, that the issue width and number of functional units are the same, and that each instruction can be run on only one type of functional unit. This thesis classifies architectures that make assumptions of this nature as idealized architectures; those that do not are considered realistic architectures.

Wilken et al. [39] evaluated the optimality of critical-path list scheduling by comparing against a highly refined integer programming solution embedded in the Gnu Compiler Collection (GCC). They tested with the SPEC95 Floating Point benchmark, which yielded basic blocks with sizes up to 1,000 when compiled with GCC's highest level of optimization. The target architecture was a single issue, fully pipelined processor, with a maximum instruction latency of 3. Their work shows that integer programming techniques alone are quite slow. As well, an integer programming approach may not scale well when more complicated architectural models are used or when multiple-issue architectures are considered. They were eventually able to schedule all basic blocks optimally, but relied on many complicated graph transformations and reductions to determine optimal schedules in place of using integer programming. Integer programming was only necessary for 0.30% of the basic blocks, those for which all other attempted methods were suboptimal or for which optimality could not be proven. They evaluated their work against a critical-path list scheduler and found that only 0.39% of basic blocks were improved over list scheduling with respect to static schedule length.

Van Beek and Wilken [38] ran another experiment using the same set of basic blocks, compiler, optimizations, and target architecture. This time, instead of using integer programming and other enhancements, the instruction scheduling problem was cast as a constraint satisfaction problem. Constraint programming led to a simpler, more efficient solution. All basic blocks were optimally scheduled, and 0.39% of basic blocks were improved over list scheduling, as in [39]. While these results seem positive, Wilken's motivation for using this target architecture was that it is the simplest architecture for which no optimal polynomial algorithm exists, so optimal methods will only have a hope of being successful on other architectures if they succeed on this one. This in turn suggests that list scheduling will produce the best schedules on this architecture, but may be increasingly outperformed by optimal methods on increasingly complex architectures.

Heffernan and Wilken [16] examine the quality of list scheduling while using graph transformations to reduce the work required for the scheduler. They evaluate using the SPEC 95, SPEC 2000, and MediaBench benchmarks, compiled by GCC with the highest level of optimization. They target single-issue, 2-issue, and 4-issue processors with a maximum latency of 4, with an even split between integer and floating point functional units on the multiple-issue architectures. For each basic block, they compute a lower bound and use critical path list scheduling. For "non-trivial" blocks where the lower bound and schedule length do not match, they employ two sets of graph transformations, using critical path list scheduling after each. They found that up to 13.2% of the non-trivial basic blocks were improved after graph transformations were used, but only a small percentage of the evaluated basic blocks were non-trivial. Their research reinforces the near-optimality of list scheduling on simple architectures.

Malik, McInnes, and van Beek [27] extend the constraint programming methods of [38] to a more complicated set of architectures. They examine four architectures, each fully pipelined, having either 1, 2, 4, or 6 functional units. The issue width on each is equal to the number of functional units, and there are up to four unique types of functional units. Instructions had a maximum latency of 38. They evaluated against a large number of basic blocks produced by IBM's TOBEY compiler, and performed instruction scheduling both before and after register allocation.

Basic blocks were scheduled using a constraint programming approach augmented with the graph transformation techniques of [16]. They used the SPEC 2000 benchmark suite, and scheduled 168,999 basic blocks before register allocation and 183,112 after register allocation. A time limit of 10 minutes was enforced, though the authors noted that decreasing the time limit to 100 seconds caused at most 10 more timeouts per architecture. With the larger time limit, the maximum number of timeouts for a single architecture was 22. They found that 98.9%-99.6% of basic blocks were scheduled optimally by list scheduling when scheduling was performed before register allocation, and 97.7%–98.8% of basic blocks were scheduled optimally when list scheduling was invoked after register allocation.

Shobaki and Wilken [36] designed the first optimal scheduler for superblocks using enumeration with several pruning techniques for efficiency. They scheduled for four fully pipelined architectures, with 1, 2, 4, or 6 functional units of up to 4 different types. The maximum latency on any instruction is 9, and the issue width on each architecture equals the number of functional units. Their test data was the majority of the SPEC 2000 benchmark suite, excluding a few benchmarks that would not compile in GCC 3.4 and those written in Fortran 90. They first evaluated the performance of three heuristics on the floating point benchmarks to determine which of the three led to the most schedules being provably optimal (having lower and upper bounds equal) before invoking their optimal schedule. Successive Retirement [6] gave the tightest bounds, performing better than DHASY [5] and critical path. In the best case, critical path yielded provably optimal schedules 92% of the time, compared to 96% for successive retirement. In the worst case, critical path was provably optimal 51% of the time, compared to 88% for successive retirement.

They selected the Successive Retirement heuristic to provide an upper bound on schedule length and compared against an optimal schedule. Their results ranged from 92.8% of blocks being scheduled optimally by Successive Retirement for a single-issue architecture to 98.3% for the architecture with 6 functional units and an issue width of 6. Compared to results presented previously for basic blocks, their experiments suggest that the critical path heuristic is designed for basic blocks and that a better-performing heuristic for superblocks must account for side exits.

3.2.2 Realistic Architectures

I now examine results concerning list scheduling using architectural models that either model a physical architecture exactly or are quite similar. Unlike the idealized architectures in the preceding section, the models mentioned here must be nearly identical to a physical architecture, though they need not be realistic in every aspect.

Ertl and Krall [10] developed an approach to instruction scheduling using constraint programming. Their methods were targeted to the Motorola 88100 processor, a multiple-issue RISC processor with a maximum latency of six. They compared their results against the schedules produced by GCC 1.37, presumably a list scheduler, and found 81% of the basic blocks to be optimal. Their benchmark suite consisted of five relatively small applications with under ten thousand operations in total between the five. While their work does not provide significant analysis on the quality of list scheduling, it does hint that list scheduling could be near-optimal.

Schielke [34] examined the optimality of list scheduling by comparing critical-path list scheduling to several stochastic scheduling techniques. Compilation was performed on a research compiler that employed a number of optimizations, none of which significantly enlarged basic blocks. Some of the scheduled basic blocks were taken from popular benchmarks, but most come from undisclosed sources. He targets three architectures, obtaining similar results for each. The Simple VLIW architecture has five functional units and full pipelining. The c60 architecture has eight fully-pipelined functional units. The PPC750 architecture has six functional units and an issue width of two, and only the integer and floating point units are fully pipelined.

Schielke's methodology begins with calculation of a lower bound for scheduling problems by combining critical path length with unavoidable hardware delays. He then uses a list scheduler to obtain an upper bound. If the two bounds match, the list scheduler must be optimal. If not, two stochastic methods described earlier in his thesis are used to search for a better solution. If none can be found, the problem is cast as an integer programming problem and the CPLEX solver is invoked with three minutes in which to find the best possible solution. If CPLEX can find an optimal solution with schedule length identical to the list scheduler's upper bound, list scheduling is clearly optimal for the given scheduling problem. If either the stochastic methods or CPLEX find better solutions, list scheduling is suboptimal, and if no method is able to produce better results or CPLEX times out, no conclusion is drawn concerning the optimality of list scheduling with respect to the given problem. Schielke finds list scheduling to be optimal with respect to static schedule length for 95–99% of all basic blocks for most benchmark programs.

Schielke's results appear to be convincing, assuming that he scheduled a representative sample of basic blocks. However, despite an explicit claim otherwise, the chosen programs seem hand-picked. He selects a few programs each from several sources instead of using a standardized benchmark suite such as the SPEC benchmark. Schielke also says nothing about the range of sizes of the basic blocks in his selected benchmark programs. Only one of the twenty-six programs has an average basic block size over 20 instructions per block, and none have more than 30 instructions per block on average. This allows for the results to be inflated by a high number of trivial basic blocks consisting of only a few instructions. Schielke claims that a few programs have some "large" basic blocks; what remains unclear is whether his chosen set is representative of basic block size in general. He also fails to examine any relationships between basic block size and the optimality of list scheduling.

Kästner and Winkel [25] examine an integer programming solution on the Intel Itanium architecture. Itaniums have 9 pipelined functional units and an issue width of 6. Each instruction type can be executed on one or more types of functional unit. The maximum latency of any instruction is 24 cycles. Kästner and Winkel solve scheduling problems optimally using a two-phase integer programming formulation, and compare its performance against list scheduling.

Their work was done in the Intel Itanium C++ compiler, and they tested against nine benchmark applications, most of them from the SPEC 95 suite. Their basic blocks were likely small in size; no exact quantities were given but the average length of basic block for each benchmark is under 10. They evaluated their methods by performing list scheduling, using a highest-level first heuristic, and comparing their schedule length against a calculated lower bound. For 87% of their basic blocks, the lower bound and schedule length matched. For the other 13%, integer programming was invoked. Overall, only 4% of their benchmark basic blocks were improved over list scheduling.

The strength of these results lies in the close correlation between their integer programming model and the actual Itanium architecture. It would seem that list scheduling performed well, even with an uncommon heuristic, for a complicated architectural model. This is mitigated by the small size of the basic blocks being scheduled, and the small total number of basic blocks, 1,787. Their results clearly attest to the near-optimality of list scheduling for schedules with few instructions, even with a complex architectural model, but the results do not necessarily scale to large basic blocks.

Liu and Chow [26] present a scheduler developed by Cognigine for the VISC architecture, an embedded processor for network hardware. VISC chips have two pairs of identical functional units, each capable of executing two operations concurrently, for an issue width of size eight. In order to represent eight operations in a 64–bit instruction, the VISC architecture uses a custom dictionary of limited size. Both the instruction schedule and the contents of the dictionary corresponding to a basic block are produced by the compiler's scheduling phase, and a feasible schedule must be small enough to fit in the dictionary in addition to respecting data and resource constraints. The architecture runs in a five-stage pipeline and has several atypical constraints placed on scheduling, such as one bank of registers that can only be accessed once per operation.

The scheduler was implemented in a variation of SGI's Pro64 compiler, and uses bounded enumeration to find a near-optimal schedule. Liu and Chow compared their scheduler to a list scheduler with a benchmark of five network application developed in-house by Cognigine. A total of 487 basic blocks were scheduled, with an overall average of 9 operations per block. The enumeration scheduler was guided by a critical path heuristic, and ties were broken by choosing the instruction involved in the most constraints and then the instruction with the highest number of successors. The list scheduler used the same heuristic as the enumeration scheduler. Liu and Chow found that their enumeration scheduler outperformed the list scheduler, producing 13.2-14.6% fewer cycles over an entire benchmark application. No metric is given to indicate the range of savings over individual basic blocks.

Of prime importance in Liu and Chow's work is their claim that list scheduling and other heuristic approaches are nearly optimal only for simple architectures, and that irregular data and resource constraints on more complex architectures cause heuristic approaches to obtain suboptimal solutions. Their results differ from those previously presented in that Liu and Chow's benchmarks are all similar programs instead of a more diverse set of benchmarks. They also do not study a large enough sample of basic blocks for their results to be accurately compared with the other presented results.

No papers have yet been published that evaluate the performance of any list scheduling heuristics for superblocks against an optimal scheduler for a realistic architecture.

3.2.3 Summary of Empirical Results

Tables 3.1 and 3.2 summarize the results discussed in this chapter. Table 3.1 displays the number of blocks and percentage of blocks solved optimally by list scheduling for each experiment mentioned in the chapter. Table 3.2 presents the relevant properties of the architectural models used in each experiment.

The work presented in this chapter leads to several conclusions. List scheduling has been shown to be near-optimal on idealized architectures for any size of basic block or superblock that appears in practice, provided an appropriate heuristic is selected. On more realistic architectures, list scheduling has proven to be near-optimal only for very small basic blocks or simple or limited benchmark suites. It is an open question, which I address in this thesis, whether or not list scheduling is near-optimal on realistic architectures for superblocks, large basic blocks, and benchmark suites that properly represent the full range of basic blocks compiled on an architecture.

Table 3.1: Results of experiments comparing list scheduling to optimal methods

Authors	Year	# Blocks	% of blocks solved optimally by list scheduling	Scheduling before register allocation?
K. Wilken, J. Liu, and M. Heffernan	2000	7,402	99.61%	No
P. van Beek and K. Wilken	2001	7,402	99.61%	No
M. Heffernan and K. Wilken	2005	Unspecified[a]	86.8%[b]	No
A. Malik, J. McInnes, and P. van Beek	2005	168,199	98.9%–99.6%	Yes
A. Malik, J. McInnes, and P. van Beek	2005	183,112	97.7%–98.8%	No
G. Shobaki and K. Wilken	2004	7,961	92.8%-98.3%	Unspecified
M. A. Ertl and A. Krall	1991	9,124	81%	No
P. Schielke - SLVIW	2000	21,496	88.2%–100.0%	Yes
P. Schielke - c60	2000	21,496	88.2%–100.0%	Yes
P. Schielke - PPC750	2000	21,496	88.2%–100.0%	Yes
D. Kästner and S. Winkel	2001	1,787	96%	Unspecified
J. Liu and F. Chow	2002	487	85.4%–86.8%	Yes

[a]The benchmarks used are SPEC 95, SPEC 2000, and MediaBench, which contain a significant number of basic blocks.

[b]This value is for non-trivial basic blocks: those not provably optimal by list scheduling.

Table 3.2: Architectural models used in experiments comparing list scheduling to optimal methods

Authors	Year	Fully Pipelined?	# Functional Units	Issue Width	Max Latency
K. Wilken, J. Liu, and M. Heffernan	2000	Yes	1	1	3
P. van Beek and K. Wilken	2001	Yes	1	1	3
M. Heffernan and K. Wilken	2005	Yes	1, 2, 4	1, 2, 4	3
A. Malik, J. McInnes, and P. van Beek	2005	Yes	1, 2, 4, 6	1, 2, 4, 6	38
G. Shobaki and K. Wilken	2004	Yes	1, 2, 4, 6	1, 2, 4, 6	9
M. A. Ertl and A. Krall	1991	No	3	3	6
P. Schielke - SLVIW	2000	Yes	5	5	20
P. Schielke - c60	2000	Yes	8	8	16
P. Schielke - PPC750	2000	No	6	2	31
D. Kästner and S. Winkel	2001	Yes	15	6	24
J. Liu and F. Chow	2002	Yes	4	8	1

Chapter 4

Local Instruction Scheduling

In this chapter, I evaluate the performance of the list scheduling algorithm against an optimal scheduler when solving the local instruction scheduling problem. Many simplifying assumptions about the target architecture are made by both the list scheduler and the optimal scheduler, and I incrementally remove assumptions from the optimal scheduler, producing a more accurate architectural model. This chapter presents the initial model and discusses each improvement made, giving a formal representation of each successive improvement. It also contains results for comparison experiments between the optimal scheduler and the initial and improved models.

4.1 Initial Model

Malik, McInnes, and van Beek [27] developed the optimal scheduler and initial model. Their scheduler, and the architectural assumptions it makes, was a starting point for this thesis.

Many simplifying assumptions are made in their initial architectural model. All instructions have an execution time of one cycle, and so the model processor is fully pipelined. The issue width is equal to the number of functional units. The model processor has an infinite number of registers and data cache misses never occur. Each instruction executes on only one type of functional unit. Lastly, each basic block has full resources available before the block begins executing, meaning that all registers are available for use in the basic block instead of containing values from a previous basic block and that no functional unit is executing any instruction when the current basic block begins execution.

Also of note is an assumption that is *not* made. There is no restriction on the number of functional units, nor is it necessary to have certain types of functional units. This provides the flexibility to model many potential architectures, such as a single-issue processor with only one functional unit or the Intel Itanium with nine functional units of four different types and an issue width of six.

The initial constraint programming model and each type of constraint used is described in the following subsections. The reader is referred to [27] for more details on the constraint programming model and optimal scheduler. All constraints that are not latency or resource constraints, such as functional unit or issue width constraints, are non-essential: their removal does not lead to an incorrect solution. However, they are added in order to reduce the domains of variables, improving performance by requiring less branching while performing backtracking search. Figure 4.1 is a DAG that has examples of every type of constraint in the constraint programming model.

4.1.1 The Constraint Programming Model

In our constraint programming model, each DAG node i has a corresponding variable X_i, for $1 \leq i \leq n$. Assigning a value d to a variable X_i means that in the schedule produced, instruction X_i will be issued in cycle d. Each variable X_i has bounds $[a_i, b_i]$, where $dom(X_i) = \{a_i, a_i + 1, \ldots, b_i - 1, b_i\}$. Our optimal scheduler uses *bounds consistency* [30] to ensure that each variable has at least one possible value that can be assigned to it. A constraint C is bounds consistent if for each variable $X_i \in vars(C)$, both a_i and b_i have a support in C. When a value is assigned to a variable, any constraints in which that variable is involved may no longer be bounds consistent. Bounds consistency can be reestablished by removing unsupported values from the domains of each variable, tightening the variable's bounds.

4.1.2 Latency Constraints

Latency constraints are binary constraints of the form $X_j \geq X_i + l(i, j)$, where $l(i, j)$ is the latency between instructions i and j. The weighted edges in the DAG translate directly into latency constraints.

Example 1 For the DAG in Figure 4.1, there are 10 latency constraints added to the constraint programming model, one for each edge in the DAG. Two such latency constraints are $B \geq A + 1$ *and* $C \geq A + 2$.

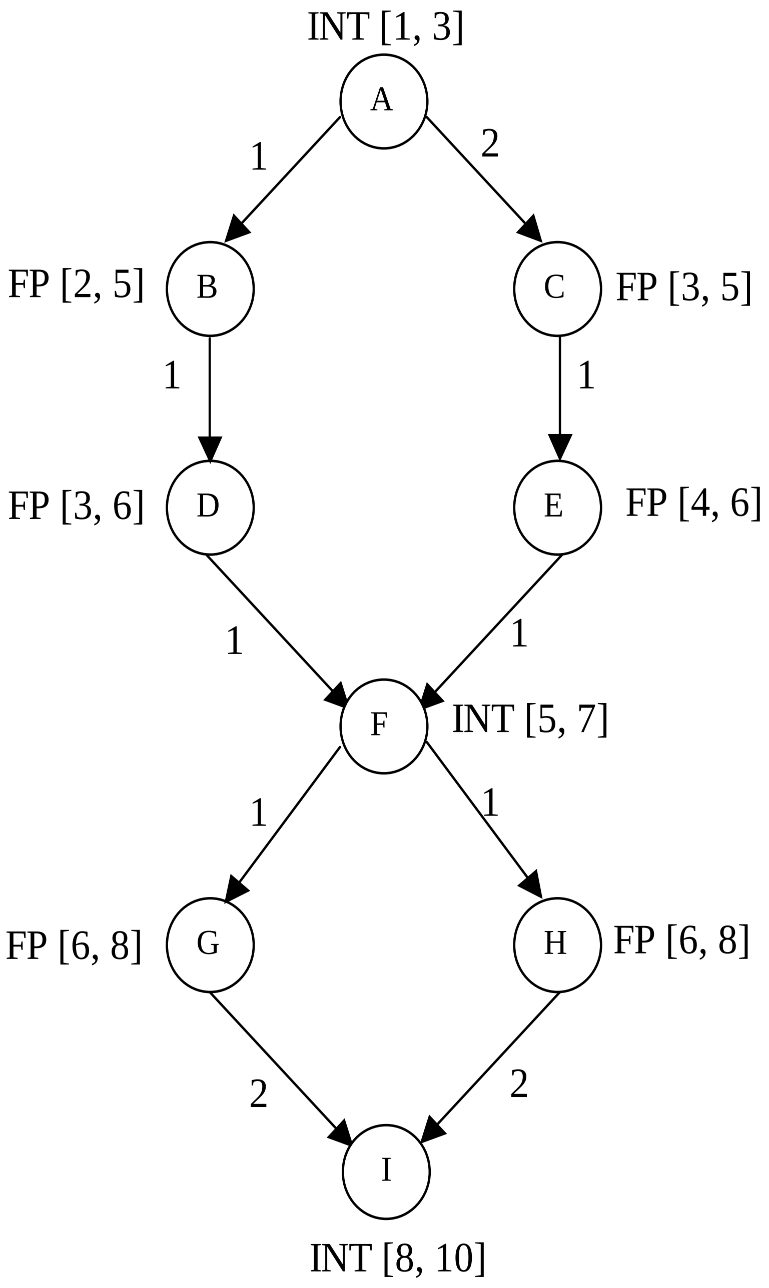


Figure 4.1: Example DAG used to illustrate constraints in the initial model. Text beside each node denotes functional unit as integer (INT) or floating point (FP) and lower and upper bounds.

4.1.3 Distance Constraints

Distance constraints are similar in nature to latency constraints but are essential to reducing the cost of backtracking search. Distance constraints, like latency constraints, are binary constraints of the form $X_j \geq X_i + d(i,j)$, where $d(i,j)$ is the "distance" between i and j. The distance between two nodes i and j is defined as the number of cycles that must pass after i has been issued before j has been issued due to resource availability. Let $onpath(i,j,t)$ be the set of instructions of type t that lie on a path between i and j, and let $r_1(i,j,t)$ be the minimum critical path distance from i to any node in $onpath(i,j,t)$. There are $f(t)$ functional units of type t, and so it will take a minimum of $r_2(i,j,t) = \lceil |onpath(i,j,t)|/f(t) \rceil$ cycles to execute those instructions, where $f(t)$ is the number of functional units of type t. Lastly, let $r_3(i,j,t)$ be the minimum critical path distance from any node in $onpath(i,j,t)$ to j. It takes at least $r_1(i,j,t)+r_2(i,j,t)+r_3(i,j,t)$ cycles to execute all instructions of type t, and so the lower bound on distance due to resources is derived, $d(i,j) = max_t\{r_1(i,j,t) + r_2(i,j,t) + r_3(i,j,t)\}$.

Example 2 *Consider the region of the DAG in Figure 4.1 with source A and sink F. There are four other nodes that lie on any path between A and F: $B, C, D,$ and E. All four nodes are floating point nodes. The minimum critical path distance from A to one of the four nodes is 1, and the minimum critical path distance from one of the four nodes to F is also 1. Assuming the target architecture has a single floating point functional unit, $d(A,F) = 1 + \lceil 4/1 \rceil + 1 = 6$, and the bounds for F are tightened from $[5,7]$ to $[6,7]$.*

4.1.4 Dominance Constraints

Heffernan and Wilken [16] introduce a set of graph transformations that reduce the cost of finding an optimal solution to the instruction scheduling problem. In our constraint programming model, we add constraints instead of performing the transformations [27]. These constraints, known as dominance constraints, are *safe*: if the constraint model was solvable before the constraints are added, it will remain solvable after the constraints are added. The following theorem, presented in [27], both defines dominance constraints and describes when it is safe to add them.

Theorem 1 (Heffernan and Wilken [16]) *Let A and B be disjoint, isomorphic subgraphs induced from the DAG with vertex sets $V(A) = \{a_1, \ldots, a_r\}$ and $V(B) = \{b_1, \ldots, b_r\}$. If,*

(a) *a_i is neither a predecessor nor a successor of $b_i, 1 \leq i \leq r$*

(b) for all predecessors k of a_i such that $k \notin V(A), l(k, a_i) \leq cp(k, b_i), 1 \leq i \leq r$

(c) for all successors k of b_i such that $k \notin V(B), l(b_i, k) \leq cp(a_i, k), 1 \leq i \leq r$

(d) for any edge $(b_i, a_j), l(b_i, a_j) \leq cp(a_i, b_j)$,

the constraints $a_i \leq b_i, 1 \leq i \leq r$ can safely be added to the constraint programming model.

Example 3 The subgraphs induced from the DAG in Figure 4.1 with vertex sets $\{B, D\}$ *and* $\{C, E\}$ are isomorphic. B is neither a predecessor nor successor of C and D is neither a predecessor nor successor of E, so condition (a) holds. B has one predecessor, A, and $l(A, B) = 1 \leq cp(A, C) = 2$, so condition (b) holds. E has one successor, F, and $l(E, F) = 1 \leq cp(D, F) = 1$, so condition (c) holds. Lastly, there are no edges from C or E to B or D, so condition (d) holds trivially. Thus, by Theorem 1, the dominance constraints $B \leq C$ and $D \leq E$ can be added to the constraint programming model.

4.1.5 Predecessor and Successor Constraints

Predecessor and successor constraints are added to the constraint programming model to estimate the distance due to resource availability between a node and its immediate predecessors or successors. I describe predecessor constraints here; successor constraints are intuitively similar.

Let $pred(i, t)$ be the set of immediate predecessors of node i that are of type t. For every type t and every subset P of $pred(i, t)$ where $|P| > f(t)$, the following constraint can be added:

$$a_i \geq min\{a_k \mid k \in P\} + \lceil |P|/f(t) \rceil - 1 + min\{l(k, i) \mid k \in P\}$$

In other words, from the earliest cycle in which any instruction in P can be issued, there must be enough cycles to execute all predecessors of i of type t in addition to accommodating the latency between any instruction in P and i.

Example 4 Consider node I from the DAG in Figure 4.1. I has two predecessors, G and H. Both are floating point instructions. Assuming one floating point functional unit, there is only one set P of size greater than one: the set $\{G, H\}$. Both G and H have a lower bound of 6, and the latency between either node and I is 2. The predecessor constraint for I is $I \geq 6 + \lceil 2/1 \rceil - 1 + 2 = 9$. The bounds for I are tightened from $[8, 10]$ to $[9, 10]$ due to this constraint.

4.1.6 Functional Unit Constraints

Functional unit constraints are global cardinality constraints, one for each unique type of functional unit. Functional unit constraints take the form gcc($X_{i1}, \ldots, X_{ik}$, $c_{v_1}, \ldots, c_{v_m}$). Each variable in the constraint is of type t, and every variable of type t must be included. There is one count variable for each cycle from 1 to some upper bound m, and the domain of each count variable is $\{0, 1, \ldots, f(t)\}$, where $f(t)$ is the number of functional units of type t. Definition 1 illustrates a functional unit constraint over a single cycle; a global cardinality constraint is merely a compact way of representing a large number of these single-cycle constraints.

Example 5 For the DAG in Figure 4.1, the functional unit constraint for integer instructions will be *gcc*$(A, F, I, c_1, \ldots, c_{10})$*, as there is an upper bound of 10 cycles on the final instruction, and thus on the schedule length. The domains* $dom(c_i)$ *of the count variables* $c_i, 1 \leq i \leq 10$ *are all* $\{0, 1, 2\}$*, assuming two integer functional units on the architecture.*

4.2 Architectural Improvements

The initial model presented in Section 4.1 is a starting point towards the development of an optimal scheduler that accurately models a physical processor. The next portion of this chapter discusses each improvement made to the initial model, describing each formally. Experimental results are presented only for the complete set of improvements to avoid any bias that might occur due to the order in which improvements were added.

4.2.1 Issue Width

On many architectures, such as the IA-64 [25] and PowerPC [19], the issue width does not equal the number of available functional units. On the IA-64, of which the Intel Itanium is an example, instructions are encoded in bundles of 128 bits containing three instructions each [18]. Two bundles can be issued simultaneously, for a total issue width of six instructions. However, the IA-64 has nine functional units.

There are several reasons for having an issue width that is smaller than the number of functional units. If the architecture is not fully pipelined, and many instructions have a execution time of more than one cycle, there is less of an advantage in being able to issue as many instructions as there are functional units, as there will often be cycles when a functional unit is already occupied and cannot accept another issued instruction. Similarly, in many blocks, various constraints on the order of instructions limits the number of instructions that can be issued simultaneously, and especially on machines with a large number of functional units, like the IA-64, there is little need to be able to issue as many instructions each cycle as there are functional units. Indeed, for even the ppc603 processor from the PowerPC family, which has only four functional units and an issue width of two, there were many basic blocks scheduled over the duration of this work for which the schedule produced has cycles in which one instruction or no instructions are issued. If a larger issue width is not needed in many cases, a processor manufacturer can reduce costs by providing a smaller issue width. Issue width also impacts the clock rate, logic complexity, and power consumption of an architecture [18].

Adding issue width constraints is a straight-forward modification to the initial model. The initial model already uses global cardinality constraints to ensure that the number of instructions of any type t issued each cycle does not exceed $f(t)$, the number of functional units of that type. To ensure that solutions are consistent with respect to the issue width, I added a global cardinality constraint involving all variables.

Definition 5 (Issue Width Constraint) $\forall k, W \geq \sum_{i=1}^{n} y_{ik}$

Example 6 The issue width constraint for the DAG in Figure 4.1 is *gcc(*$A, \ldots, I,$ $c_1, \ldots, c_{10}$*)*, as there is an upper bound of 10 cycles on the final instruction, and thus on the schedule length. If the target architectural model has an issue width of 4, the domains $dom(c_i)$ of the count variables $c_i, 1 \leq i \leq 10$ are all $\{0, \ldots, 4\}$.

4.2.2 Non-Fully Pipelined Processor

Almost all modern architectures are not fully pipelined, including the Intel Pentium and Itanium and the PowerPC architectures. A fully pipelined architecture requires every instruction to have an execution time of 1, which is only worthwhile if there is very little variance, if any, in required processing time for each instruction in the instruction set. On most architectures, some instructions require significantly more processing time than the majority. For example, compare integer addition with floating point square root on the PowerPC 604: addition takes one cycle while square root takes 32. In order for every instruction to have an execution time of one cycle, the cycle time would have to be long enough for the longest instruction in the architecture to complete. In each cycle where one of those long instructions was not scheduled, computing power would be wasted. Suppose for simplicity that the PowerPC ran at a frequency of one cycle per second, and so addition takes one second to complete and square root takes 32 seconds. If the PowerPC 604 was converted to a fully pipelined architecture, each cycle would require 32 seconds of time, and it would take 32 seconds to perform integer addition when only one second was required.

To be more efficient, most architectures are not fully pipelined, and so there will be cycles in which instructions cannot be issued on a particular functional unit, since the unit will still be executing a previously-issued instruction. To model this feature, I introduce *pipeline variables*, special variables that are added to the CSP.

Definition 6 (Pipeline Variables) *Suppose an instruction i with corresponding CSP variable X_i has bounds $[a_i, b_i]$ and execution time $e(i)$, with $e(i) > 1$. Insert variables $p_{i,j}$ into the CSP, for $1 \leq j \leq e(i) - 1$. Each variable $p_{i,j}$ is of functional unit type $u(i)$. $p_{i,j}$ has bounds $[a_i + j, b_i + j]$. Also add all pipeline variables of type t to the functional unit constraint for type t (see Section 4.1.6).*

Since the bounds of $p_{i,j}$ are related to the bounds of X_i, this forces the pipeline variables to be scheduled in sequence following X_i. In particular, when $a_i = b_i$, $p_{i,j}$ has bounds $[a_i + j, a_i + j]$, and a pipeline variable for instruction X_i is forced to execute in the next $e(i) - 1$ cycles following the cycle in which X_i was issued.

While I do not augment the DAG with additional nodes in practice, another way to conceptualize pipeline variables is that each corresponds to an imaginary node added to the DAG, forming a chain of nodes with edge weights of 1 between the initial node y that is not fully pipelined and the final pipeline node z. In this scenario, there need be no edge between z and any successor of y. Some successors of y have edges with weights much less than the execution time of y since some DAG edges do not correspond to read-after-write dependencies, so adding an edge between z and some successors would unnecessarily lengthen the schedule produced.

Example 7 The DAG in Figure 4.2 has nodes B_1 and B_2 added to illustrate the use of pipeline variables. Suppose the target architectural model has one floating point functional unit and that instruction B is scheduled in cycle 2. As nodes B_1 and B_2 correspond to pipeline variables, they must be issued on the floating point unit in cycles 3 and 4, and instruction C cannot be issued until instruction 5.

In every model, the latencies used were those in the data, not those in the table of instructions found in [19]. In many cases, successors of variables corresponding

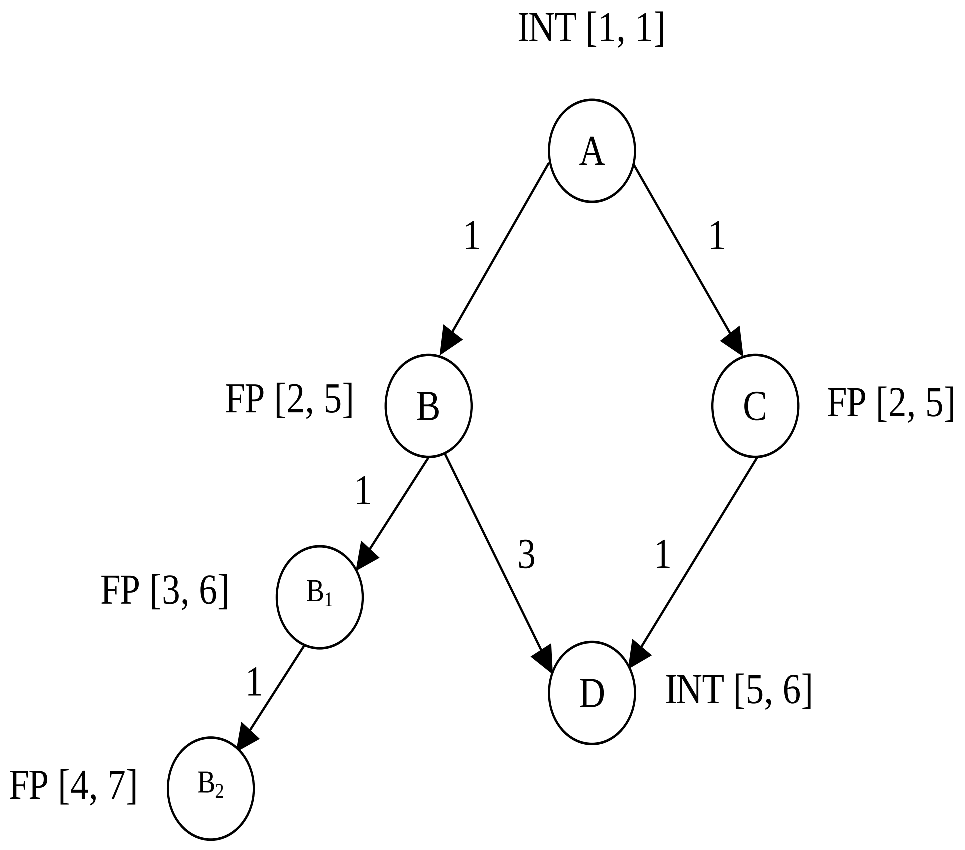


Figure 4.2: Example DAG with additional nodes B_1 and B_2 corresponding to pipeline variables.

to multi-cycle instructions had latencies that would prevent the successors from executing until the multi-cycle instruction completed execution. Reasons why the latencies would be less than the multi-cycle instructions' execution time were discussed previously in Section 2.2. The main difference between the initial model and the non-fully pipelined model is that in the initial model, an instruction could be issued the cycle after a multi-cycle instruction was issued on the same unit as the multi-cycle instruction. In the non-fully pipelined model, this cannot be done, as any instructions issued on the same unit as the multi-cycle instruction before the multi-cycle instruction completes would conflict with one of the pipeline variables. The pipeline variables effectually lock a functional unit for the full execution time of a multi-cycle instruction.

There are two major ways in which pipeline variables differ from regular variables. Pipeline variables can never be selected as the branching variable in the backtracking algorithm. Allowing the selection of a pipeline variable would increase the number of candidate branching variables with no particular benefit, as the bounds of pipeline variables are fixed in relation to the bounds of the root variables. Pipeline variables are also not counted towards the issue width, since they correspond to a previously-issued instruction that is executing in some cycle after the issue cycle, not a new instruction being issued.

Pipeline variables are considered when global cardinality constraints on the number of available functional units of one type are propagated, as they contribute towards the availability of the functional units.

4.2.3 Serializing Instructions

While performing instruction scheduling, the schedule must always be equivalent to a serial schedule produced by the compiler's instruction generation phase. A *serial schedule* is a schedule in which only one instruction is issued per cycle and no two instructions execute at the same time. Compilers produce serial schedules in the code generation phase, and most, if not all, high-level programming languages are serial languages, with the exception of those that allow concurrency. Even then, individual concurrent units, such as threads, are executed in a serial order, with one high-level instruction following another.

Instruction scheduling may produce non-serial schedules when scheduling for architectures with an issue width greater than one, but the behaviour of the scheduled code must be exactly equivalent to the behaviour of a serial schedule. Access to architectural resources may force a schedule to be partially serialized, or for only one instruction to be issued in a given cycle if the instruction has certain properties. For example, architectures may have only one of a particular resource, such as the condition register on the PowerPC 604 [21], and need to ensure that only one instruction is accessing that resource at a time. Some architectures may enforce instruction ordering on the processor, by stalling some instructions until it is safe to issue them. Other architectures, including the PowerPC [19, 21] and Intel Pentium [24] and Itanium [22], require order to be enforced by the compiler, either by providing instructions that serialize the processor or by creating a serial-friendly schedule.

The PowerPC Compiler Writer's Guide [19] describes four types of instructions that require some sort of serialization and occur on the PowerPC architecture. However, only one of these types occurs with any frequency in our test data (see Section 4.3.1), occurring 15% of the time compared to less than 1% for each of the other three types, and instructions with the same properties can also be found on the Intel Pentium [24] and Itanium [22] architectures. This type of serializing instruction, labelled *execution serialization* in PowerPC literature [19, 21], describes a set of instructions that require exclusive access to the processor in the cycle in which they are issued. These instructions are held in a queue on the processor until they are the oldest uncompleted instruction on the processor (in other words, until all previously executing instructions have completed), and then they are issued. In the cycle in which they are issued, no other instruction can be issued, meaning that for one cycle, the instruction has sole access to the processor and its resources. Instructions having these exact properties will be referred to as *serializing instructions* throughout this thesis.

Serializing instructions can be modelled in a CSP in a manner similar to that of instructions with a execution time larger than one, as described in Section 4.2.2. I introduce a *serial variable*, which is similar to a pipeline variable.

Definition 7 (Serial Variables) Suppose an instruction i with corresponding CSP variable X_i has bounds $[a_i, b_i]$ and represents a serial instruction. Insert variables $s_{i,j}$ into the CSP, for $1 \leq j \leq F-1$, where F is the total number of functional units. There is one serial variable for every functional unit except for the one on which instruction i is issued; the functional unit type of each serial variable is assigned accordingly. $s_{i,j}$ has bounds $[a_i, b_i]$. Also add all serial variables of type t to the functional unit constraint for type t (see Section 4.1.6).

Since the bounds of $s_{i,j}$ are exactly equal to the bounds of X_i, this forces the serial variables to be issued in the same cycle as X_i. As there are enough serial variables to fully occupy every functional unit except for one to be used for X_i, no

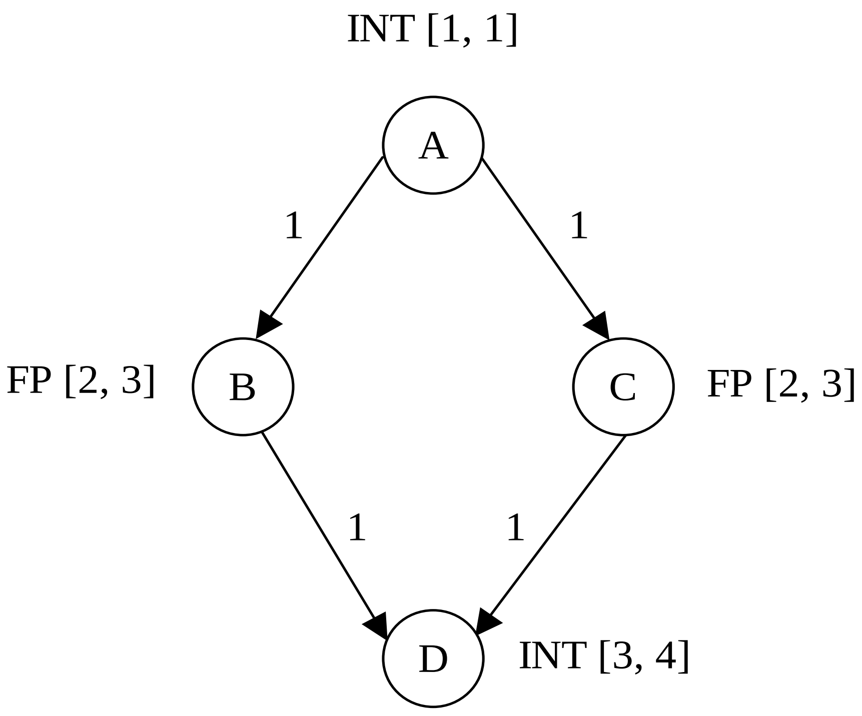


Figure 4.3: Example DAG with serial instruction B.

other instruction besides X_i can be issued in the cycle in which X_i is issued. This also disallows any multi-cycle instruction from being issued before X_i is issued but completing after X_i is issued, since the pipeline variable that would be required to be scheduled in the same cycle as X_i's issue cycle would conflict with some serial variable $s_{i,k}$. Thus, no instruction will be executing on the processor in the same cycle in which X_i is issued, and any instruction issued before X_i must have completed before X_i is issued.

Example 8 Consider the DAG in Figure 4.3, where node B corresponds to a serial instruction. Suppose the target architectural model has an issue width of 2. If neither B nor C corresponded to serial instructions, they could both be issued in cycle 2, allowing D to be issued in cycle 3. However, since B corresponds to a serial instruction, no other instruction can be issued in the same cycle as B. Thus, if B is issued in cycle 2, C must be issued in cycle 3 and D must be issued in cycle 4.

As with pipeline variables, serial variables are not selected as the branching variable in the backtracking algorithm. They are not considered in the issue width constraint as there may be more functional units than the issue width and adding all serial variables to the issue width constraint in such a case makes the CSP unsolvable.

4.2.4 Architectural Features Not Modelled

There are several other features of modern architectures that were rejected for addition to the improved model. As mentioned in 4.2.3, the three other types of serialization on the PowerPC were ignored because of the extremely low frequency of their occurrence in our benchmark suites.

Some architectures, like the IA-64 [23], allow some instructions to be executed on several different types of functional unit. For example, the **add** instruction can be executed on either an I-unit or M-unit, generally used for integer and memory instructions respectively. This capability was not added to our model as there was no accurate way to evaluate it. The PowerPC architecture does not allow instructions to execute on multiple types of functional unit. If I were to have selected some instructions from the PowerPC instruction set to allow to be issued on multiple functional unit types, I would have had to choose an arbitrary set of instructions to do so.

Basic blocks rarely have the full resources of the processor available to them when they begin execution. In particular, non-fully pipelined instructions may still be executing and registers may already be storing data from previous basic blocks. Neither of these features were modelled because of the difficulty of finding a suitable evaluation method. While I could have designated some set of registers and functional units to be in use at the start of a basic block, this would be another arbitrary choice.

Cache misses were not modelled as the contents of the data cache while a basic block is executing do not depend solely on the basic block. Without a compiler that considered all basic blocks at once, it would be impossible to predict cache usage during basic block scheduling.

4.3 Evaluation

I now present data which evaluates the performance of list schedulers when compared to our optimal scheduler. The data is collected across a large number of basic blocks, nearly 400,000, each scheduled on several architectures, for the initial and improved architectural models described previously in this chapter.

4.3.1 Experimental Setup

To evaluate the quality of our optimal scheduler compared against list scheduling in both the initial and improved architectural models, I scheduled basic blocks for a number of architectures using both the list and optimal schedulers, and compared the schedule lengths. All experiments were done on a 3.2 GHz Intel Pentium machine with 1 GB of RAM.

The basic blocks used in all experiments were the complete SPEC 2000 benchmark suite. The SPEC benchmark suite is a collection of C, C++, and Fortran programs that is widely used in research and production in compilers and architectures for experimental evaluations. The basic block data was generated by IBM's TOBEY compiler backend, and includes basic blocks generated both before and after register allocation was performed.

I compared the optimal scheduler against list scheduling using two heuristics: Critical Path and Shieh and Papachristou, both described fully in Section 2.4.1. For each model improvement and heuristic, all basic blocks were scheduled on each of four architectural models, differing in the issue width and number and type of functional units only. Properties such as the execution time of each instruction were obtained from the PowerPC Compiler Writer's Guide [19]. Table 4.1 summarizes the differences between each of the models used. For every experiment, including those simulating architectures other than the PowerPC family (in particular, the Intel Itanium), the PowerPC instruction set was used, due to the fact that our research group had ready access to basic blocks compiled on the PowerPC and did not have access to basic blocks compiled on the Itanium. We are thus mapping PowerPC instructions to generic versions of the functional units available on an Itanium.

Changes were made to the list scheduler only for the purposes of facilitating evaluation. The aim of this thesis was to evaluate the quality of schedule produced by a standard list scheduler on both idealized and realistic architectural models, as list schedulers are used in production compilers. It therefore makes little sense to improve the list scheduler except for evaluation purposes, as described later in this chapter.

There are several possible ways in which the list and optimal schedulers could be compared. One possibility is for actual schedules to be generated, compiled, and compared with respect to overall runtime on a standard set of input data. This does not necessarily reflect schedule quality, as the optimal scheduler could produce significantly better schedules for most blocks but the list scheduler could produce optimal results for the basic blocks in an application that are executed

Table 4.1: Architectural models used in scheduling experiments

Architecture	Issue Width	Simple Int. Units	Complex Int. Units	Memory Units	Branch Units	Floating Point Units
1-Issue	1	1				
PowerPC 603e (ppc603e)	2	1		1	1	1
PowerPC 604 (ppc604)	4	2	1	1	1	1
Intel Itanium (IA-64)	6	2		2	3	2

most frequently. Another disadvantage is that the selection of test input might bias the behaviour of the application, and thus favour the schedules produced by one of the two schedulers.

Another alternative is to run the code in a PowerPC simulator, such as Mambo [4] or PSIM, which measures the actual cycles expended, including delays for cache misses, and compare the cycle count. This approach suffers from the same drawbacks as actually timing native code discussed above.

The final method of evaluation is to compare the lengths of schedules produced by the optimal and list schedulers. While this is the simplest method of evaluation, it fails to accurately represent the running time of a particular program, and it does not clearly indicate how well frequently-executed basic blocks perform as we have no profiling information and thus cannot accurately estimate the actual time savings.

I chose the final method because it avoids the dependency of the results on the choice of input data and the variability that different input data could cause, and because this method of evaluation is the most common in papers that compare heuristic and optimal scheduling methods. While this choice of evaluation does not account for cache misses, there is no obvious reason why either scheduler should consistently have worse cache performance than the other, so they are ignored, especially since this evaluation method does not measure cache performance in any way.

As the architectural model was improved, comparing static schedule lengths with the initial list scheduler became ineffective. The list scheduler produced schedules that ignored and even violated the advanced architectural features, allowing it to produce shorter schedules than the optimal scheduler. For example, on a single-issue machine with a 32-cycle floating point instruction, the list scheduler would still schedule an instruction every cycle, if possible, while an optimal scheduler using a non-fully pipelined model, would be forced to delay 32 cycles before any other instruction could be issued.

Because of this drawback, the list scheduler was modified for evaluation purposes. As list scheduling occurs, special conditions are enforced so that the *choices* the list scheduler makes are exactly as they would be when scheduling for our initial model, but once a choice of instruction is made, the scheduler *behaves* as if it was a physical processor, enforcing delays as necessary. If a schedule produced by the initial list scheduler were executed on a processor, the processor would delay instructions when resources were unavailable; the modified list scheduler described later in this chapter copies that behaviour. The 32-cycle instruction on the single-issue machine would be chosen without any knowledge of its execution time or impact on other instructions beyond what the basic list scheduler would already know, but once this instruction is chosen, the functional unit executing the instruction is locked for the full 32 cycles, and the list scheduler cannot successfully select another instruction until the 32-cycle instruction completes.

4.3.2 The Optimal Scheduler

The goal of the optimal scheduler is to search the space of all schedules having length in the range $[L, U-1]$, attempting to find the shortest possible schedule with length at least L and at most U for lower and upper bounds L and U. This is done by searching first for a feasible schedule of length L. If none can be found, it searches for a scheduler of length $L+1$, continuing until it searches for a schedule of length $U-1$. The scheduler need not attempt to find a schedule of length U, provided that U is equal to the schedule length of some not necessarily optimal schedule, such as one generated by a list scheduler using any heuristic. If there is no valid schedule with length at most $U-1$, and a heuristic has produced a schedule of length U, it follows that U is the length of the optimal schedule, and the optimal scheduler has proved the heuristically-generated schedule to be optimal.

The lower bound L is calculated by initializing the lower bounds of all variables to 1 and then modifying lower bounds so that latency and distance constraints are satisfied. For each variable in topologically sorted order, the latency, distance, predecessor, and successor constraints are propagated, tightening the lower bound of each variable as much as possible. A similar process is used to calculate the initial upper bounds, save that the order in which variables are processed is reversed and that the upper bounds of all variables are initialized to L.

The upper bound U is obtained by invoking the list scheduler on the DAG. To achieve the best possible bound, I take the minimum schedule produced by using several different heuristics. In addition to the Critical Path and Shieh and Papachristou heuristics, the optimal scheduler also considers a Decision Tree heuristic, learned by a decision tree algorithm from data obtained from our optimal scheduler [33], an Earliest Start Time heuristic and two refinements to Shieh and Papachristou using more tie breaking features that proved in practice to have a negligible effect on schedule length (and thus were not selected to be one of the heuristics used by the list scheduler). The Earliest Start Time heuristic selects the instruction with the earliest start time, breaking ties with the critical path to sink.

Having set initial bounds on the variables and determined an upper bound on schedule length, the optimal scheduler can now search for an optimal schedule. If $L = U$, the schedule produced by the list scheduler is provably optimal and no further work need be done. If not, the scheduler tries to find a schedule of length L. If no feasible schedule can be found, the upper bounds on all variables are increased by one, the source and sink have their bounds set to $[1, 1]$ and $[L+1, L+1]$ respectively, and the scheduler attempts to find a schedule of length $L + 1$. This alternation between increasing bounds and searching for a schedule of length k continues until either a schedule of length k is found or the scheduler determines that no schedule with length $k \leq U - 1$ exists.

The optimal scheduler searches for a schedule of length k using a recursive backtracking search interleaved with constraint propagation. At each level, a variable X_i with bounds $[a_i, b_i]$ is chosen heuristically to be the branching variable. Variable X_i is assigned the value a_i, and constraints are propagated. The propagation algorithm first propagates any constraints involving X_i, and notes which variables have their bounds changed. Constraints involving this new set of changes variables are propagated, resulting in another changed set. Propagation continues until no variables have their bounds changed as a result of constraints in the most current set of constraints, or until any variable has its domain reduced to the empty domain. If propagation completes with a valid partial assignment, a new branching variable is selected.

If a variable has its domain reduced to the empty domain, there is no solution to the CSP given the current partial assignment. This means that the value assigned to X_i is invalid. The domains of all variables are restored to their values before X_i was assigned a_i, and the lower and upper bounds of X_i are set to $a_i + 1$. This new change is then propagated. X_i is assigned each value in $[a_i, b_i]$ successively, followed by propagation, until either a solution is found or the scheduler determines that for X_i assigned to any value in $[a_i, b_i]$, no solution exists. A solution is found when propagation completes successfully and there are no variables left with unequal lower and upper bounds. For a DAG with n nodes, there need not be n levels of propagation, both because the sink and source are assigned values and because propagating constraints when one variable is assigned may lead to other variables being assigned.

If there is no value within $[a_i, b_i]$ which, when assigned to X_i, leads to a solution, the algorithm *backtracks*. The partial solution for all variables assigned values before X_i was selected as the branching variable cannot lead to a solution, and so the value assigned to the previous branching variable is invalid. It is increased, if possible, and propagation is performed; if this previous variable was also assigned its upper bound, the algorithm backtracks again, and so on. If X_i was the first branching variable, there is no solution given the current constraints and schedule length, and so the schedule length is increased as described previously.

The optimal scheduler uses several different levels of propagation that allow for a tradeoff between the time spent propagating constraints and the quality of the propagation that gets performed. Latency, distance, edge, and functional unit constraints must be propagated at every level of propagation, as failure to do so could lead to an invalid schedule being identified as valid. Singleton consistency, a propagation technique discussed later in this chapter, helps restrict the bounds of the variables, but is not necessary for correctness. For basic blocks with few nodes which can be scheduled quite quickly, the cost of performing singleton consistency outweighs the speed increase gained by fewer backtracks and variable assignments being required. For this reason, the scheduler uses three levels of propagation, using higher levels of propagation as the time spent scheduling increases.

Initially, singleton consistency is not performed. If the scheduler fails to find an optimal solution within 5 seconds, the scheduler begins using singleton consistency to depth one. If no solution is found within 15 seconds, the scheduler uses singleton consistency to depth two. If no solution is found after 10 minutes, the scheduler times out, and the schedule length returned is a lower bound to the optimal schedule length. The time limit of 10 minutes was chosen as a balance between time and number of blocks that could be solved without timing out; very few additional blocks could be solved if the time limit was increased to 30 minutes.

Branching variables are chosen heuristically from the set of variables for which the lower and upper bounds are not currently equal. The heuristic used varies slightly depending on the propagation level. If the lowest level of propagation is being used, all variables are considered. If not, let ch_i be the number of changes in the domains of variables which occurred when singleton consistency was propagated for variable X_i, and let $ch = \max\{ch_i\}$. In this case, variables are candidates for the next branching variable if $ch_i \geq 0.8 * ch$, favouring the instruction that has the greatest effect on the domains of other variables. There must always be at least one instruction for which $ch_i \geq 0.8 * ch$, as $ch = ch_j$ for some j and $ch_j \geq 0.8 * ch_j$. The value 0.8 was chosen as it produced good results in practice, being large enough to reduce the number of variables considered but small enough to allow there to often be some choice of variable.

Singleton Consistency

Singleton consistency (see [7]) is an additional form of constraint propagation that can, in practice, significantly reduce the domains of variables and shorten the amount of time needed to find an optimal solution to a constraint satisfaction problem. Singleton consistency is based on the property that for variable X_i, a value $v \in dom(X_i)$ that has a support in a constraint C will have a support in C even if $dom(X_i) = \{v\}$. Singleton consistency works by temporarily reducing the domain of a variable to a singleton set and propagating constraints to find an inconsistency. If an inconsistency is found, the value is removed from the domain of the variable.

In our scheduler, we place a bound on how many levels of recursion are used during singleton consistency for performance reasons. If no limit is placed on the number of levels of recursion for which singleton consistency is performed, singleton consistency becomes another form of backtracking search. Since we are enforcing bounds consistency within our scheduler, we choose the values for the singleton domains for a variable by starting at the variable's lower bound. If an inconsistency is detected, the lower bound is increased by one and a new singleton domain is made until there is no inconsistency within the given depth of propagation. A similar method is used to reduce the upper bound on a variable. We perform singleton consistency on all variables which do not already have a singleton domain.

4.3.3 Results for Initial Architectural Models

The list scheduler performs quite well for the initial model. As Table 4.2 and Table 4.3 indicate, there are very few blocks for which the list scheduler is not optimal. The list scheduler produces optimal schedules for 99.1%-99.9% blocks when scheduling was done before register allocation and 98.6%-99.6% blocks when scheduling was done after register allocation. Tables 4.4 and 4.5 present the same results grouped by block size instead of benchmark application. However, when the optimal scheduler is able to find an improved schedule, the schedule is often improved by a significant number of cycles. Table 4.6 and Table 4.7 show the average and maximum percentage improvement over all improved basic blocks, with a maximum improvement over any block on any architecture of 27%. While the optimal scheduler is able to produce significantly better schedules when the heuristic schedule is non-optimal, there are so few non-optimal schedules produced that the overall savings are negligible.

Comparing the four architectures, the worst performance for the list scheduler is achieved on the PowerPC 603e and 604 architectures; the single-issue and IA-64 architectures have fewer improved schedules and the improvements are generally smaller than on the PowerPC architectures. In the case of the single-issue architecture, I suspect that the optimal scheduler is unable to find many better schedules than the list scheduler because there is often a clear best choice of instruction to issue in a given cycle, and both heuristics are able to find this best instruction. While a single-issue architecture can be the least forgiving of a poor choice (on a multi-issue architecture, if instruction a is the best choice and b is second-best, the list scheduler can choose b and then a, but both may be executed in the same cycle; on a single-issue architecture, a single bad choice of instruction can cause the schedule to be non-optimal), the small issue width also makes the best instruction more obvious.

The performance of the list scheduler on the IA-64 is harder to explain. Perhaps since the issue width and number of functional units are both large, the need to make the right choices in a heuristic is reduced, since if an instruction must be executed in a particular cycle in order to lead to an optimal schedule, there are more chances for that instruction to actually be selected by the heuristic than there are on a more constrained architecture. The PowerPC architectures are complicated enough to often have more than one strong choice of instruction, as opposed to the single-issue architecture, but have small enough issue widths to make wrong choices costly and are less forgiving than the IA-64.

There is also negligible difference between the two heuristics used. Table 4.8 and Table 4.9 shows that there is little gain realized when choosing one heuristic over the other, although Shieh and Papachristou's heuristic does outperform the critical path heuristic.

List scheduling performs well with respect to register pressure. When scheduling is performed before register allocation, the list scheduler gets better results on every architecture, as shown in Table 4.10. After register allocation, the list scheduler produces schedules with a lower register pressure more often than the optimal scheduler for the PowerPC 603e and IA-64 architectures, and is not significantly outperformed by the optimal scheduler on the single-issue and PowerPC 604 architectures (see Table 4.11). With respect to the two list scheduling heuristics, Shieh and Papachristou's heuristic outperforms the critical path heuristic when scheduling is done before register allocation, as shown in Table 4.12, but critical path produces better results after register allocation, demonstrated in Table 4.13.

Register pressure is more significant when instruction scheduling is performed before register allocation instead of after register allocation. This is because register allocation assigns a physical register to each data value, possibly at the cost of introducing spill code to temporarily store some values in memory, and so a schedule can never use more registers than are physically available [29].

The numbers presented in Tables 4.10 and 4.11 only show results for basic blocks where the schedule produced by the list scheduler was not proved optimal. When the list scheduler does give a provably optimal schedule, the optimal scheduler is either not invoked or stops after determining that there is no valid schedule of length $U-1$, where U is the upper bound given by the list scheduler. Since only the list scheduler produces an actual schedule in this case, there can be no comparison between schedules.

Overall, list scheduling performs quite well, and there is no strong motivation to use an optimal scheduler for an idealized architectural model. This is not surprising, as it is supported by the research surveyed in Chapter 3. The list scheduler is outperformed by the optimal scheduler mainly for larger blocks, but with the majority of basic blocks being small, this is not particularly significant.

Table 4.2: *Local instruction scheduling for the initial architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3128	30		29		26		2	
applu	653	18		23		22		6	1
apsi	2210	16		65		70		15	
art	355			1				4	
bzip2	972	4		2		1			
crafty	4969	41		26		26		7	
eon	4509	22		25	1	12	1	8	
equake	486	1		4		3		2	
facerec	1221	12		48		54		16	
fma3d	10034	133		207	9	143	8	39	11
galgel	5369	55		90	3	81	2	13	2
gap	19729	47		41		17		5	
gcc	42686	107		81		38		25	
gzip	1610	11		20		13			
lucas	915	15		49		51		16	
mcf	364	9		8		2			
mesa	14903	87		177	3	130	3	42	
mgrid	207	1		3		4			
parser	3561	8		3		2			
perl	16450	47		59		29		8	
sixtrack	10950	213		333	2	284	2	38	2
swim	345	1		4		4		1	1
twolf	7468	55		45		35		3	
vortex	11945	54		82		30		7	
vpr	3369	18		15		6			
wupwise	591	12		14		12		1	
Totals	168999	1017		1436	18	1079	16	241	17

Table 4.3: *Local instruction scheduling for the initial architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3459	73		55		52		7	
applu	734	33		47		47		16	
apsi	2650	57		103		100		38	
art	486	1		4				4	
bzip2	1060	5		17		13			
crafty	5135	56	1	82	1	58		21	
eon	4972	87		107		74		26	
equake	503	5		7		7		3	
facerec	1436	21		65		61		31	
fma3d	11280	333	2	432	2	288	1	118	3
galgel	6120	139		195		183		43	
gap	20625	151		64		33		16	
gcc	45565	174		147		77		30	
gzip	1722	10		18		14		1	
lucas	1014	21		33		35		9	
mcf	407	10		9		2			
mesa	16478	177		199	1	125	1	44	
mgrid	221	8		15		12		3	
parser	3934	19		6		4		1	
perl	17542	144		102		74		10	
sixtrack	12568	402		645		584		195	3
swim	388	6		6		6		1	
twolf	7695	70		71		35		12	
vortex	12808	60		126		70		41	
vpr	3654	26		30		17		4	
wupwise	654	41		50		45		9	
Totals	183110	2126	3	2631	4	2014	2	677	6

Table 4.4: *Local instruction scheduling for the initial architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of basic blocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	91806	50	0.1	52	0.1	43	0.0	0	0.0
6-10	44152	257	0.6	243	0.6	112	0.3	3	0.0
11-20	20167	246	1.2	334	1.7	187	0.9	41	0.2
21-30	5321	137	2.6	201	3.8	158	3.0	37	0.7
31-50	3895	161	4.1	262	6.7	230	5.9	50	1.3
51-100	2372	124	5.2	234	9.9	238	10.0	49	2.1
101-250	1131	36	3.2	89	7.9	94	8.4	48	4.3
251-2600	155	6	3.9	21	14.4	17	11.6	14	9.9
Totals	168199	1017	0.6	1436	0.9	1079	0.6	241	0.1

Table 4.5: *Local instruction scheduling for the initial architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of basic blocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	90305	51	0.1	48	0.1	34	0.0	0	0.0
6-10	47655	242	0.5	203	0.4	105	0.2	6	0.0
11-20	25766	6203	2.3	602	2.3	324	1.3	89	0.3
21-30	8446	333	3.9	451	5.3	307	3.6	95	1.1
31-50	5808	365	6.3	503	8.7	471	8.1	168	2.9
51-100	3273	302	9.2	485	14.8	466	14.2	182	5.6
101-250	1655	181	10.9	279	16.9	257	15.5	113	6.8
251-2600	199	49	24.6	60	29.9	58	24.9	24	12.2
Totals	183107	2126	1.2	2631	1.4	2014	1.1	677	0.4

Table 4.6: *Local instruction scheduling for the initial architectural model before register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	8.8	12.5	9.5	16.7	8.8	10.0	0.0	0.0
6-10	7.5	11.1	8.7	14.3	8.3	16.7	11.5	16.7
11-20	5.3	16.7	6.9	21.1	6.8	16.7	8.2	12.5
21-30	3.2	10.7	4.9	21.1	5.0	15.0	6.4	10.0
31-50	2.4	13.2	3.8	17.2	3.6	15.0	4.4	10.5
51-100	1.9	8.2	2.3	26.9	2.6	22.2	3.8	20.7
101-250	1.5	8.8	2.8	21.4	3.0	21.4	2.1	17.6
251-2600	0.3	0.7	2.5	12.4	3.6	12.4	1.7	9.2
Totals	4.7	16.7	5.4	26.9	4.8	22.2	4.7	20.7

Table 4.7: *Local instruction scheduling for the initial architectural model after register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	9.2	14.3	9.6	16.7	9.1	10.0	0.0	0.0
6-10	7.3	11.1	8.4	14.3	7.7	14.3	11.3	16.7
11-20	4.6	13.3	6.0	21.4	5.4	12.5	7.7	12.5
21-30	3.1	11.1	4.6	16.7	4.5	13.3	5.3	11.5
31-50	2.3	10.8	3.4	23.5	3.5	15.0	4.3	14.3
51-100	1.8	8.1	2.5	17.1	2.9	21.4	3.2	16.7
101-250	1.2	7.8	1.4	10.0	1.5	11.7	1.5	8.6
251-2600	0.2	0.6	0.5	3.2	0.5	3.6	0.5	2.7
Totals	3.6	14.3	4.3	23.5	3.8	21.4	4.1	16.7

Table 4.8: *Local instruction scheduling for the initial architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over Shieh and Papachristou's heuristic, and (b) Shieh and Papachristou's heuristic resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	156	0	57	0	6	0	0
6-10	23	121	19	70	11	30	6	4
11-20	76	101	78	125	53	68	10	82
21-30	19	67	47	73	46	41	17	22
31-50	25	32	48	76	45	60	11	15
51-100	15	51	29	74	22	76	21	40
101-250	12	29	20	56	20	53	16	27
251-2600	2	1	3	13	3	12	7	9
Totals	177	558	244	544	200	346	88	199

Table 4.9: *Local instruction scheduling for the initial architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over Shieh and Papachristou's heuristic, and (b) Shieh and Papachristou's heuristic resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	4	85	0	17	0	8	0	0
6-10	28	196	47	154	36	51	10	3
11-20	177	197	170	233	115	99	56	39
21-30	88	92	111	123	89	86	40	45
31-50	148	107	150	232	137	209	80	48
51-100	166	91	188	187	157	184	59	100
101-250	90	54	103	106	93	90	34	54
251-2600	18	21	22	24	21	23	4	11
Totals	719	843	791	1076	648	750	283	300

Table 4.10: *Local instruction scheduling for the initial architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	15	103	1	30	1	7	0	5
6-10	567	609	191	488	96	118	28	62
11-20	1284	1185	760	783	369	446	303	367
21-30	217	221	450	522	320	337	281	258
31-50	597	864	440	423	288	290	207	215
51-100	309	617	220	396	211	290	175	200
101-250	92	420	133	185	136	158	109	115
251-2600	12	63	21	35	21	34	11	39
Totals	3093	4082	2216	2862	1442	1680	1114	1222

Table 4.11: *Local instruction scheduling for the initial architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	71	148	22	60	12	15	0	12
6-10	707	537	314	502	162	196	30	154
11-20	1654	1322	1032	930	820	507	375	460
21-30	982	885	492	340	622	381	435	352
31-50	847	759	556	515	473	437	343	365
51-100	354	449	230	410	241	315	183	255
101-250	160	237	101	146	96	139	101	64
251-2600	27	34	20	22	21	23	20	23
Totals	4802	4371	3010	3143	2445	2017	1475	1730

Table 4.12: *Local instruction scheduling for the initial architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than Shieh and Papachristou's schedule, and (b) Shieh and Papachristou's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	201	94	190	30	18	7	17	5
6-10	333	244	401	273	145	87	131	55
11-20	358	549	412	525	291	329	237	312
21-30	217	221	239	300	173	248	154	204
31-50	109	228	177	212	158	135	129	107
51-100	80	173	112	204	108	162	130	103
101-250	34	189	31	114	26	105	56	51
251-2600	2	34	9	29	8	23	16	17
Totals	1334	1732	1571	1687	927	1093	870	854

Table 4.13: *Local instruction scheduling for the initial architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than Shieh and Papachristou's schedule, and (b) Shieh and Papachristou's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	143	138	125	60	50	15	47	12
6-10	412	191	384	303	227	153	189	127
11-20	575	456	732	598	650	342	425	289
21-30	408	302	492	340	436	207	294	255
31-50	335	284	470	313	405	232	298	237
51-100	215	162	244	222	236	191	207	161
101-250	99	134	121	61	118	58	101	64
251-2600	20	11	15	12	19	13	15	18
Totals	2207	1678	2583	1909	2141	1043	1576	1163

4.3.4 Results for Improved Architectural Models

As the initial model was improved by the addition of more realistic architectural properties, it became necessary to modify the list scheduler in order to accurately compare schedule lengths. This section discusses the modifications made to the list scheduler and presents the results of comparing the performance of both schedulers when the optimal scheduler was given an improved architectural model.

List Scheduler Modifications

In order to produce the behaviour in the list scheduler described in Section 4.3.1, the list scheduler had to undergo several changes, described here. Our initial list scheduler did not account for issue width since the initial CSP model did not either. Issue width is a standard feature of modern list schedulers, and so this change was made directly.

The list scheduler simulates an *in-order processor*, one which executes instructions in the order specified by the program. The alternative would be an *out-of-order processor*, which does not necessarily execute instructions in the order in which they are received by the processor. The two processor classes differ mainly in their behaviour when an instruction is issued too early. For example, on an architecture with one integer and one floating point functional unit, no integer instructions can be issued for five cycles following the issue of an integer instruction i with an execution time of five. Suppose the next two instructions to be issued are instructions j, an integer instruction, and k, a floating point instruction. An in-order processor would delay both j and k until five cycles after i had been issued, while an out-of-order processor could issue k before i had finished executing, provided it could guarantee the correctness of the program. I chose to modify the list scheduler to resemble an in-order processor, not an out-of-order processor, because the former is more clear and it allows for a direct comparison against the optimal scheduler. If the list scheduler resembled an out-of-order processor, any comparison would be lost unless the optimal scheduler behaved the same way. Given the NP-completeness of optimal instruction scheduling, incorporating out-of-order behaviour into a constraint program will drastically increase the runtime of the optimal scheduler without adding any value to this thesis.

To evaluate list scheduling on a processor that was not fully pipelined, the list scheduler was modified to simulate the processors' behaviour when attempting to issue an instruction for which no functional unit was available. If a compiler produced code that attempted to issue instructions without having proper units available, the processor would delay those instructions (and any dependent instructions) until an appropriate functional unit became available. This effect is achieved in a standard list scheduler by having a *pending list*: a list of instructions that were selected to be issued in the current cycle but cannot be issued because a functional unit is unavailable. When choosing an instruction to schedule, instructions **must** be chosen from the pending list if possible, and the oldest instruction of its type for which a functional unit is available must be chosen first. For example, suppose the pending list has two floating point instructions and an integer instruction, with the integer instruction between the floating point instructions in order of time. The integer instruction can be chosen as soon as an integer unit becomes available, but the newer floating point instruction cannot be chosen until the older floating point instruction has been chosen. If the pending list is empty or if no instructions in the pending list can be executed but another instruction can, that instruction may be issued. When choosing an instruction that is not on the pending list, **selectBestInstruction** does not consider the execution time of the instruction; the heuristics used remain the same as they were previously.

The effect is that **selectBestInstruction** makes a choice that is uninformed with respect to execution time and available functional units, just as in a standard list scheduler, and the pending list simulates the delay enforced by the processor if no functional units are available for a particular instruction.

Instructions which serialize the processor are also not a feature of the standard list scheduling algorithm. The necessary modifications were similar to those for non-fully pipelined processors. If a serial instruction is issued in the current cycle, no other instructions can be issued. If **selectBestInstruction** selects a serial instruction but another instruction has been issued in the current cycle or a previously-issued instruction is executing, the serial instruction is placed on the pending list. If any serial instructions are on the pending list, no other instruction can be issued unless it completes without delaying the first serial instruction on the pending list. For example, if a 32-cycle floating point instruction has been issued and then a serial instruction is selected and added to the pending list, a 3-cycle load instruction can be issued, as it does not delay the serial instruction (provided that it finishes before or at the same time as the floating point instruction), as this does not further delay the execution of the serial instruction. Only the first serial instruction need be considered: since only one serial instruction can execute in a cycle, delaying the first serial instruction necessarily delays all future serial instructions. As before, **selectBestInstruction** has no knowledge of the pending list; it selects an instruction from the ready list using a standard heuristic. This simulates the delay experienced when a serial instruction is issued on the processor but cannot execute until other instructions complete execution. As with non-fully pipelined processors, an instruction on the pending list must be selected, if there is one, to prevent the list scheduler from being able to decide that the previously queued instruction was not the best choice after all.

In the case where there are both serial instructions and instructions delayed due to multi-cycle instructions, the oldest instruction is selected first, regardless of type. The only exception is that an instruction waiting on a multi-cycle instruction may be selected ahead of an older serial instruction provided that the serial instruction is not further delayed, similar to the example in the preceding paragraph.

Results

While Section 4.2 presents architectural improvements in succession, the results here encompass all implemented architectural improvements. There is no motivation to choose one improvement before another for the purposes of evaluation, and so all are evaluated at once. The order presented in Section 4.2 follows the order of implementation and testing, which has no bearing on the performance of either scheduler when all improvements are considered.

The list scheduler's performance is still quite good when using the improved architectural model. Tables 4.14 and 4.15 give the improvements by benchmark application, and Tables 4.16 and 4.17 give the improvements by block size. List scheduling is now optimal from 95.9%-97.8% of the time when scheduling is performed before register allocation and 94.2%-97.6% of the time when scheduling is performed after register allocation. When the list scheduler is not optimal, it is further away from optimality for the improved architectural model than the initial architectural model. Table 4.18 and Table 4.19 give average and maximum improvements over all improved basic blocks. In the worst case, the optimal scheduler finds a schedule nearly half as long as the list scheduler (the maximum percentage improvement of 42.7% on the IA-64 architecture when scheduling is done after register allocation).

With regard to heuristics, Shieh and Papachristou's heuristic achieves much better schedules than the critical path heuristic (see Tables 4.20 and 4.21).

The list scheduler and optimal scheduler are again relatively similar with regards to register pressure. Neither significantly outperforms the other for any target architecture, and there seems to be little difference whether scheduling is performed before or after register allocation. Tables 4.22 and 4.23 give results comparing the two schedulers, and Tables 4.24 and 4.25 compare the critical path heuristic with Sheih and Papachristou's heuristic. As with the initial architectural model, Shieh and Papachristou's heuristic produces better schedules with regards to register pressure than the critical path heuristic when scheduling is done before register allocation, but critical path performs better after register allocation.

Overall, list scheduling is still quite good, both in terms of schedule length and register usage. The optimal scheduler produces better results for large basic blocks, but the overwhelming number of small basic blocks is a good argument in favour of list scheduling.

Table 4.14: *Local instruction scheduling for the improved architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3128	129	6	146	9	137	10	101	26
applu	653	64	1	110	8	102	7	71	13
apsi	2210	169	28	338	45	349	27	259	13
art	355	1		15		16		9	
bzip2	972	7		28		27	3	24	1
crafty	4969	57		136	3	171	9	137	9
eon	4509	137	1	378	22	374	17	310	34
equake	486	6		11	1	10	1	8	3
facerec	1221	64		138	22	155	10	145	11
fma3d	10034	958	6	1114	72	769	56	697	101
galgel	5369	143	1	413	9	452	11	358	13
gap	19729	452	1	680	1	477	1	448	1
gcc	42686	144	2	398	5	374	4	390	8
gzip	1610	11		44		42		33	1
lucas	915	47		68		55		41	
mcf	364	9		17		16	2	15	1
mesa	14903	353		709	23	632	12	875	27
mgrid	207	3		11		13		12	
parser	3561	20		43		27		36	
perl	16450	57		320		279	1	278	9
sixtrack	10950	662	15	1376	33	1125	29	808	81
swim	345	11		34	4	32	1	27	10
twolf	7468	131		325		219	1	184	1
vortex	11945	94		291	2	222	4	195	5
vpr	3369	75		101	2	73	3	59	4
wupwise	591	27	1	48		47		36	
Totals	168999	3769	62	6964	264	5996	209	5195	372

Table 4.15: *Local instruction scheduling for the improved architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	3459	135	6	208	12	208	10	158	21
applu	734	84	3	138	10	146	12	118	20
apsi	2650	179	12	455	41	477	27	398	24
art	486	3		26		20	1	30	
bzip2	1060	13		62		63	4	56	6
crafty	5135	81	2	218	11	233	12	209	19
eon	4972	173	2	605	97	539	24	560	47
equake	503	9		15	3	21	2	18	3
facerec	1436	71		231	31	211	22	201	16
fma3d	11280	951	13	1517	87	1241	93	1277	186
galgel	6120	219		633	16	710	20	602	18
gap	20625	515		736	2	680	10	684	19
gcc	45565	212	1	1052	6	1221	18	1245	24
gzip	1722	11		74	1	82	1	80	4
lucas	1014	49		158	1	158	1	143	2
mcf	407	10		25	1	28	1	24	1
mesa	16478	378	3	1054	29	1044	31	1132	52
mgrid	221	13		30	1	31	1	23	1
parser	3934	30		146		155	1	161	1
perl	17542	154		669	1	694	5	782	16
sixtrack	12568	780	17	1806	51	1636	47	1259	109
swim	388	24		52	1	45	3	45	9
twolf	7695	145		325		298	3	258	7
vortex	12808	90		446	6	580	5	564	16
vpr	3654	88	2	189	3	180	4	185	8
wupwise	654	59		93	1	96	1	72	2
Totals	183110	4416	61	10571	412	10455	359	9675	631

Table 4.16: *Local instruction scheduling for the improved architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of basic blocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	91806	70	0.1	722	0.8	682	0.7	692	0.8
6-10	44152	560	1.3	1073	2.4	771	1.7	810	1.8
11-20	20167	1217	6.0	1774	8.8	1452	7.2	1095	5.4
21-30	5321	630	11.8	1138	21.4	1070	20.1	887	16.7
31-50	3895	640	16.4	1067	27.6	976	25.2	838	21.7
51-100	2372	448	19.1	849	37.1	729	31.6	582	25.7
101-250	1131	179	16.2	309	29.5	287	27.3	280	27.6
251-2600	155	25	16.3	34	31.8	31	27.7	12	15.2
Totals	168999	3769	2.2	6964	4.1	5996	3.5	5195	3.1

Table 4.17: *Local instruction scheduling for the improved architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of basic blocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	90305	64	0.1	502	0.6	403	0.4	406	0.5
6-10	47655	488	1.0	980	2.1	851	1.8	859	1.8
11-20	25766	1338	5.2	3061	11.9	2932	11.4	2760	10.7
21-30	8446	702	8.3	1843	21.9	2120	25.1	1913	22.7
31-50	5808	818	14.1	2060	35.7	2072	36.1	1783	31.3
51-100	3273	602	18.5	1367	43.4	1341	42.6	1195	38.7
101-250	1655	336	20.6	693	46.4	666	42.7	718	49.6
251-2600	199	68	37.8	79	61.2	81	58.3	44	48.9
Totals	183107	4416	2.4	10571	5.8	10455	5.7	9675	5.3

Table 4.18: *Local instruction scheduling for the improved architectural model before register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	8.0	12.5	15.8	25.0	16.2	20.0	16.5	25.0
6-10	6.7	18.2	9.0	26.7	9.6	26.7	10.2	31.3
11-20	5.5	21.7	7.1	26.3	7.7	27.8	8.7	31.0
21-30	5.6	23.8	6.3	30.8	6.3	32.5	7.4	33.8
31-50	4.8	23.2	4.6	26.5	5.4	32.4	6.5	37.1
51-100	3.6	29.3	3.4	33.3	3.7	31.9	4.4	28.8
101-250	3.4	23.5	3.1	20.4	3.6	21.1	3.7	16.8
251-2600	1.1	13.3	2.6	7.8	1.7	10.1	0.9	3.3
Totals	5.2	29.3	7.1	33.3	7.6	32.5	8.6	37.1

Table 4.19: *Local instruction scheduling for the improved architectural model after register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the basic blocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	8.9	14.3	15.8	25.0	16.9	20.0	17.3	25.0
6-10	6.7	14.3	9.3	26.7	10.0	26.7	10.0	31.3
11-20	5.2	22.2	7.9	34.1	8.7	34.1	10.0	42.7
21-30	4.4	21.7	6.0	27.6	6.6	28.6	8.3	33.3
31-50	4.0	25.0	4.9	26.3	5.3	36.4	6.2	41.3
51-100	3.2	32.0	3.8	30.8	4.1	30.8	5.0	36.4
101-250	2.2	18.3	3.2	26.8	3.3	29.6	4.3	24.3
251-2600	1.7	17.2	1.9	11.7	2.5	15.0	3.2	10.7
Totals	4.5	32.0	6.6	34.1	7.0	36.4	8.2	42.7

Table 4.20: *Local instruction scheduling for the improved architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over Shieh and Papachristou's heuristic, and (b) Shieh and Papachristou's heuristic resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	36	156	44	159	3	62	3	63
6-10	40	116	60	242	34	241	36	181
11-20	113	104	146	463	158	436	109	293
21-30	27	89	156	216	122	207	123	152
31-50	41	33	87	223	87	229	80	101
51-100	23	62	87	134	89	138	106	90
101-250	24	24	38	104	43	102	50	41
251-2600	2	1	9	9	11	5	6	4
Totals	306	585	627	1550	547	1420	513	925

Table 4.21: *Local instruction scheduling for the improved architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over Shieh and Papachristou's heuristic, and (b) Shieh and Papachristou's heuristic resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	35	85	32	126	2	70	2	69
6-10	40	184	89	361	73	210	57	151
11-20	203	200	353	612	305	533	258	440
21-30	90	89	225	402	282	415	439	360
31-50	164	113	330	458	334	447	296	286
51-100	188	97	322	299	274	301	193	212
101-250	86	57	173	172	167	228	145	127
251-2600	17	24	18	22	20	26	8	12
Totals	823	849	1542	2452	1457	2230	1398	1657

Table 4.22: *Local instruction scheduling for the improved architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	81	386	126	275	133	238	148	239
6-10	40	116	400	989	343	335	331	326
11-20	1379	1106	1134	1280	842	820	755	686
21-30	781	747	692	625	502	500	615	463
31-50	671	720	659	527	524	369	420	316
51-100	384	529	412	392	392	338	305	253
101-250	155	309	227	134	203	122	144	108
251-2600	35	40	22	18	25	15	12	4
Totals	3526	3953	3672	4240	2964	2737	2730	2395

Table 4.23: *Local instruction scheduling for the improved architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	272	427	255	306	268	260	270	252
6-10	968	847	753	898	584	489	573	490
11-20	1859	1276	1491	1596	1430	1363	1394	1260
21-30	1046	876	935	834	862	970	864	925
31-50	880	695	765	816	702	753	627	705
51-100	456	382	380	455	325	420	283	407
101-250	204	199	158	191	172	225	156	176
251-2600	29	20	22	11	16	23	4	13
Totals	5174	4722	4759	5107	4359	4503	4171	4228

Table 4.24: *Local instruction scheduling for the improved architectural model before register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than Shieh and Papachristou's schedule, and (b) Shieh and Papachristou's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	221	92	224	56	26	40	19	41
6-10	331	252	404	301	159	102	134	79
11-20	352	535	405	607	345	325	257	283
21-30	213	216	270	307	199	236	191	199
31-50	118	202	204	238	157	152	120	147
51-100	83	170	112	197	94	174	109	87
101-250	37	156	38	87	39	81	45	38
251-2600	7	22	7	17	5	13	5	4
Totals	1362	1645	1664	1810	1024	1123	880	878

Table 4.25: *Local instruction scheduling for the improved architectural model after register allocation.* Number of basic blocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than Shieh and Papachristou's schedule, and (b) Shieh and Papachristou's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	147	135	140	92	47	48	32	51
6-10	390	169	409	317	260	182	215	157
11-20	473	482	620	683	529	389	377	331
21-30	395	328	422	347	395	308	309	248
31-50	329	284	441	342	371	254	271	258
51-100	199	151	215	214	193	185	163	179
101-250	89	122	110	90	100	73	61	87
251-2600	18	10	12	5	13	8	8	6
Totals	2040	1681	2369	2090	1908	1447	1436	1317

4.4 Summary of Results

Basic block scheduling has made heavy use of a critical path heuristic-based list scheduler for many years, and based on the work presented in this thesis, there is minimal need for that to change. The data presented in this chapter is for all basic blocks within the SPEC 2000 benchmark suite of length three or greater. When trivial blocks of length one and two are also considered, list scheduling is able to schedule even more blocks optimally overall.

There are two major findings in this chapter. The first is that while list scheduling for a more realistic architectural model produces fewer optimal schedules than for an idealized architectural model, there is not a significant loss of quality when scheduling for a more realistic architectural model. Thus, the list scheduling algorithm continues to be a worthwhile method of performing instruction scheduling.

The second finding is that while the critical path feature may be the most popular and best-performing feature in a heuristic (a fact not proven by experiments over the course of my work), the choice of secondary features is indeed significant. Shieh and Papachristou's heuristic outperformed the critical path heuristic, and as both have critical path as their primary feature, the obvious conclusion is that secondary features are indeed important. This finding suggests the need for a more through examination of heuristic performance on a sufficiently large benchmark suite, such as the work done by Russell et al. [33].

Chapter 5

Global Instruction Scheduling

Global instruction scheduling is similar in many ways to local instruction scheduling. While our optimal scheduler has many additions designed to reduce the cost of finding an optimal schedule for a superblock, the list scheduling algorithm itself is unchanged. In this chapter, I present the minor differences between local and global instruction scheduling, discuss the changes made to the optimal scheduler, and evaluate list scheduling for the initial and improved architectural models presented in Chapter 4.

5.1 Initial Model

There is only one difference in the initial model between local and global instruction scheduling. The cost function for global instruction scheduling is given in Definition 3, and it obviously differs from schedule length, the cost function used for local instruction scheduling. Other than the cost function, the model remains the same. A superblock can be scheduled by treating it as a basic block and not giving side exits special treatment. In fact, this is precisely how list scheduling works for superblocks. Unless the heuristic used treats side exits as special instructions, there are no other differences between local and global instruction scheduling with respect to the list scheduler.

5.2 Architectural Improvements

Because global and local instruction scheduling are so similar, there was no need to add any more improvements to the architectural model, nor did any existing improvements have to be changed. The issue width, for example, makes no distinction between side exits and other instructions. None of the improvements directly involve the schedule cost, and as schedule cost is the only change between local and global instruction scheduling, none of the improvements made in Section 4.2 need any modifications.

5.3 Evaluation

In the remainder of this chapter, I present experimental results gained from scheduling nearly 200,000 superblocks. As in Chapter 4, the data was obtained from compiling the entire SPEC 2000 benchmark suite in IBM's TOBEY compiler backend. Superblocks were collected before instruction scheduling was performed, both before and after register allocation were performed. Each superblock was scheduled on several different architectures using both the initial and improved architectural models. The same set of architectures given in Table 4.1 was used again for global instruction scheduling.

5.3.1 Experimental Setup

The experimental setup for global instruction scheduling differs in a few ways from the experimental setup from local instruction scheduling. The process followed by the optimal scheduler was changed in several ways to enable it to find solutions to the global instruction scheduling problem more quickly. These changes are described in Section 5.3.2. The experiments were also performed on a Linux cluster instead of a standard PC so that the results could be obtained faster. Experiments were run on Whale, an HP Opteron cluster that is part of the Shared Hierarchical Academic Research Computing Network (SHARCNET: http://www.sharcnet.ca). The Whale cluster is comprised of 768 machines running HP Linux XC 3.0, each with 4 GB of RAM and 4 2.2 GHz processors for a total of 3,072 processors.

The other more significant difference between the local and global instruction scheduling experiments is the heuristics used for evaluation. The critical path heuristic used for local instruction scheduling is used again for global instruction scheduling, but instead of using Shieh and Papachristou's heuristic, DHASY is used. This is because DHASY was found to produce optimal schedules more often than other global instruction scheduling heuristics in work done both within our research group and by others (see Section 2.4.1).

5.3.2 The Optimal Scheduler

Because the cost function for global instruction scheduling is more complex than the schedule length, as used in local instruction scheduling, the optimal scheduler must account for this. This section outlines the differences in how the optimal scheduler performs scheduling for superblocks as compared to basic blocks.

As with local instruction scheduling, the optimal scheduler invokes a list scheduler on the superblock multiple times, using a large number of heuristics. In addition to the heuristics mentioned in Section 4.3.2, two other heuristics were used. Side Exits First favours side exits, breaking ties with critical path distance and then earliest start time. The other heuristic, Weighted Estimate, is described in [8] and is similar to DHASY except that estimates of the distances between pairs of nodes that account for resource usage are used in place of the actual critical path distance.

Heuristic bounds are used differently when scheduling superblocks. Each heuristic schedule has both a length and a cost, which provide upper bounds for the optimal schedule length and optimal schedule cost. For global instruction scheduling, the optimal schedule length is the minimum length of any possible schedule, ignoring the cost function completely.

The upper bounds on schedule length and cost can be improved by considering articulation nodes. An *articulation node* is a side exit such that if it is removed from a DAG G, G will have two separate components. That is, there is no edge from a node in a basic block preceding the side exit to a node in a basic block following the side exit.

Articulation nodes are useful in tightening the upper bounds on schedule length and cost, and may even lead to an optimal solution. The optimal scheduler considers each articulation node in the DAG in the order of topological sort. It solves the region consisting of all nodes lying on a path between the initial node and the articulation node exactly, unless a time limit of 5 seconds is exceeded. This tightens the lower bounds on each articulation node in succession. If every side exit is an articulation node, it can be proven (see [28]) that a schedule with optimal cost can be obtained by scheduling each articulation node as early as possible. Since the lower bounds on each articulation node have been tightened, all that remains is to schedule while minimizing schedule length, as is done for basic blocks, and an optimal schedule is obtained. If not every side exit is an articulation node, the bounds on the articulation nodes can still be tightened.

The next step is to calculate a lower bound on the length of the optimal cost schedule. This can be found by obtaining the optimal length schedule ignoring schedule cost; in other words, by treating the superblock like a basic block. Let the optimal schedule length be L. The optimal cost schedule must have length at least L, as no schedule, optimal or not, with length less than L exists.

An upper bound on the final exit node for an optimal cost schedule can be obtained from an upper bound U on schedule cost. Schedule cost for any schedule is obtained by weighting the bounds on each exit. If every side exit is scheduled for its earliest starting time and these weighted times are subtracted from U, the remaining value is the weighted upper bound for the final exit. The final exit cannot have a higher upper bound, or else the weighted sum of the exits would exceed U. If any side exit was scheduled later than its lower bound, the final exit would have to be scheduled earlier in order for the weighted sum of the exits to be U. This upper bound on the final exit is also an upper bound on schedule length for an optimal cost schedule. The optimal scheduler then uses singleton consistency to prune the bounds of each exit variable.

At this point, the scheduler has obtained tight bounds on the cost variables and on schedule cost, so it begins a process of enumerating assignments to cost variables. For each schedule cost from lower to upper, the scheduler enumerates through all valid assignments of the exits that achieve that schedule cost from lowest to highest. For each assignment, all exits are fixed and backtracking interleaved with constraint propagation is used to determine if a schedule exists. Three levels of timeouts are used in the same way as they were used for local instruction scheduling. The second and third levels include singleton consistency to depth one and two respectively as part of each constraint propagation phase.

5.3.3 Results for Initial Architectural Models

Global instruction scheduling is clearly a more difficult problem to solve than local instruction scheduling. For the initial architectural model, Tables 5.1 and 5.2 summarize the number of blocks in which the list scheduler was non-optimal grouped by benchmark application, and Tables 5.3 and 5.4 give the number and percentage of non-optimal blocks grouped by block size. When instruction scheduling is performed before register allocation, 91.2%-97.5% of schedules produced by the list scheduler are optimal. When scheduling follows register allocation, the list scheduler produces optimal schedule for 96.3%-97.6% of the superblocks.

The average percentage improvements for improved schedules is fairly small, ranging from 3.9%-6.4% when instruction scheduling precedes register allocation and 3.2%-5.7% when scheduling follows register allocation, as shown in Tables 5.5 and 5.6. However, the maximum improvements are significant, again affirming that the list scheduler making a non-optimal choice can be costly.

Table 5.7 and Table 5.8 show, unsurprisingly, that DHASY is a much better heuristic for global instruction scheduling than critical path. In the worst case, the single-issue architecture when scheduling is performed before register allocation, critical path outperforms DHASY on 1,677 superblocks, but DHASY produces better schedules for 13,668 superblocks. This confirms that when scheduling superblocks, a heuristic that accounts for side exits is essential for good performance.

The optimal scheduler produces reasonable schedules in terms of register pressure, although a heuristic schedule produces schedules with a lower register pressure than the optimal scheduler more often than the optimal schedule has a lower register pressure than a heuristic schedule. Tables 5.9 and 5.10 show the number of basic blocks where the schedule produced by the optimal scheduler and a heuristic schedule had differing register pressures. A more surprising result is that for each architecture, when one of the heuristics produced a schedule with lower register pressure than the other, it was most often DHASY, as shown in Tables 5.11 and 5.12. This is significant as it suggests that an improved schedule need not occur at the expense of increased register pressure.

In summary, global instruction scheduling is a more complex problem than local instruction scheduling, and even on an idealized architectural model, list scheduling does not perform as well for global instruction scheduling as it did for local instruction scheduling. However, when using a heuristic designed for global instruction scheduling, the list scheduler is still near optimal for the majority of superblocks, and an optimal scheduler need not be used for most purposes.

5.3.4 Results for Improved Architectural Models

As mentioned in Section 5.2, there are no new changes needed to the improved architectural model to accommodate global instruction scheduling. As such, the list scheduler remains the same, other than the improvements presented in Section 4.3.4 that are necessary to simulate the behaviour of the processor within the scheduler.

Scheduling superblocks effectively appears to be a much more difficult for a realistic architectural model than for an idealized architectural model. Tables 5.13 and 5.14 summarize the number of blocks in which the list scheduler was non-optimal grouped by benchmark application, and Tables 5.15 and 5.16 give the number and percentage of non-optimal blocks grouped by block size. Unlike schedules produced for the idealized architectural model, a significant number of heuristic schedules are non-optimal for the realistic architectural model: 46.4%-49.6% when scheduling is performed before register allocation and 39.9%-52.3% when scheduling is performed after register allocation.

While the average percentage improvement for any architectural model is not particularly large, ranging from 5.3%-7.6% when instruction scheduling precedes register allocation and 5.1%-8.1% when instruction scheduling follows register allocation, there is a significant overall increase in schedule cost over the optimal scheduler due to the large number of blocks which are non-optimal. Table 5.17 and Table 5.18 show average and maximum improvements for schedules where the list scheduler produced a non-optimal schedule.

Table 5.19 and Table 5.20 confirm again that DHASY is a much better heuristic for global instruction scheduling than the critical path heuristic. The single-issue architecture is once again the architecture on which DHASY outperforms critical path by the widest margin: critical path finds better schedules than DHASY for 1,373 superblocks but DHASY finds better schedules than critical path for 13,400 superblocks.

When considering register pressure, heuristic schedules have lower register pressure than optimal schedules more often than optimal schedules have lower register pressure than heuristic schedules, shown in Tables 5.21 and 5.22. However, the margin is fairly narrow, with a difference between the two quantities of at most 1,400, occurring on the single-issue architecture when instruction scheduling is performed before register allocation. As with the idealized architectural model, DHASY schedules had a lower register pressure more often than critical path schedules. Tables 5.23 and 5.24 show the differences between the two heuristics with respect to register pressure.

These results show that list scheduling is far from optimal when scheduling superblocks for a realistic architectural model. Not only are 40%-52% of schedules produced by the list scheduler not optimal but the optimal scheduler yields an average improvement of 5% or more with significant improvements on some superblocks compared to the best heuristic schedule. Differences between heuristics and the optimal scheduler with respect to register pressure are insignificant.

Table 5.1: *Global instruction scheduling for the initial architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	1834	125		71		69		55	
applu	262	31		18		11		20	
apsi	1039	120		116		64		52	
art	295	23		13		6		4	
bzip2	480	41		27		15		21	
crafty	2469	323		180		111		117	
eon	2453	354		488		494		144	
equake	190	6		9		9			
facerec	580	67		68		38		33	
fma3d	4888	460		282		204		174	
galgel	2657	235		168		81		102	
gap	11518	1012		582		135		138	
gcc	24698	2280		889		544		519	
gzip	794	109		58		33		17	
lucas	552	23		32		17		4	
mcf	202	18		9		1		4	
mesa	7003	573		342		207		176	
mgrid	72	11		8		5		5	
parser	2042	127		67		41		42	
perl	9219	751		298		214		204	
sixtrack	4930	497		223		121		200	
swim	192	14		8		2		2	
twolf	4269	240		136		84		77	
vortex	6613	436		153		135		110	
vpr	1677	142		91		52		56	
wupwise	268	22		9		1		9	
Totals	91196	8038	0	4345	0	2694	0	2285	0

Table 5.2: *Global instruction scheduling for the initial architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	1950	104		94		94		58	
applu	266	25		27		28		36	
apsi	1281	60		106		99		100	
art	448	11		14		9		14	
bzip2	495	17		8		12		12	
crafty	2630	131		141		100		101	
eon	3059	117		154		138		72	
equake	195	5		8		7		7	
facerec	643	43		86		69		52	
fma3d	6009	323		328		306		278	
galgel	3046	80		168		157		140	
gap	11524	317		238		129		93	
gcc	25053	806		546		394		275	
gzip	810	44		24		30		16	
lucas	545	20		21		25		10	
mcf	236	8		7		6		7	
mesa	7775	236		273		200		215	
mgrid	78	5		13		12		9	
parser	2050	42		26		22		21	
perl	9387	302		180		157		99	
sixtrack	5505	420		481		440		377	
swim	190	5		19		20		21	
twolf	4330	119		110		70		85	
vortex	6601	214		252		218		166	
vpr	1752	55		57		46		50	
wupwise	280	7		11		6		18	
Totals	96138	3516	0	3392	0	2794	0	2332	0

Table 5.3: *Global instruction scheduling for the initial architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of superblocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	15764	171	1.1	11	0.1	3	0.0	2	0.0
6-10	22720	1357	6.0	705	3.1	675	20.0	455	2.0
11-20	27596	2510	9.1	1366	5.0	749	2.7	762	2.8
21-30	10985	1569	14.3	753	6.9	319	2.9	332	3.0
31-50	8409	1343	16.0	764	9.1	387	4.6	354	4.2
51-100	4287	840	19.6	514	12.0	360	8.4	230	5.4
101-250	1274	213	16.7	194	15.2	167	13.1	111	8.7
251-2600	161	37	23.0	38	23.6	34	21.1	39	24.2
Totals	91196	8038	8.8	4345	4.8	2694	3.0	2285	2.5

Table 5.4: *Global instruction scheduling for the initial architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of superblocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	16274	12	0.1	0	0.0	4	0.0	10	0.1
6-10	23726	251	1.1	133	0.6	180	0.8	351	1.5
11-20	28713	1056	3.7	697	2.4	571	2.0	651	2.3
21-30	11566	569	4.9	513	4.4	357	3.1	271	2.3
31-50	9123	672	7.4	734	8.0	549	6.0	363	4.0
51-100	5137	578	11.3	831	16.2	705	13.7	399	7.8
101-250	1401	308	22.0	396	28.3	351	25.1	230	16.4
251-2600	198	70	35.4	88	44.4	77	38.9	57	28.8
Totals	96138	3516	3.7	3392	3.5	2794	2.9	2332	2.4

Table 5.5: *Global instruction scheduling for the initial architectural model before register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the superblocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	13.5	20.0	16.7	16.7	16.7	16.7	16.7	16.7
6-10	7.2	18.8	6.8	17.9	9.9	20.0	15.0	33.3
11-20	4.2	22.1	4.6	23.1	6.2	22.6	6.3	24.3
21-30	2.9	34.9	3.3	27.1	4.1	16.7	3.9	17.2
31-50	2.2	29.2	2.6	19.6	2.8	18.4	2.7	16.7
51-100	1.7	34.8	1.9	25.0	2.3	25.0	2.1	21.8
101-250	1.2	16.6	1.4	15.3	1.6	10.8	1.4	10.8
251-2600	0.7	2.9	1.8	11.4	1.4	10.0	1.4	8.2
Totals	4.0	34.9	3.9	27.1	5.5	25.0	6.4	33.3

Table 5.6: *Global instruction scheduling for the initial architectural model after register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the superblocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	16.1	16.7	0.0	0.0	25.0	33.3	22.0	33.3
6-10	8.3	17.6	8.0	25.0	13.5	24.8	14.2	33.3
11-20	4.8	18.8	5.5	17.2	6.7	20.0	6.9	31.0
21-30	3.1	16.0	3.5	23.8	3.6	16.7	4.0	15.5
31-50	1.9	11.2	2.8	20.0	2.7	20.0	3.2	15.9
51-100	1.3	10.8	1.8	12.0	2.0	11.8	2.4	18.2
101-250	0.8	6.3	1.3	14.5	1.4	8.9	1.5	10.6
251-2600	0.4	1.9	0.6	4.7	0.7	5.0	0.6	3.5
Totals	3.2	18.8	3.2	25.0	3.9	33.3	5.7	33.3

Table 5.7: *Global instruction scheduling for the initial architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over DHASY, and (b) DHASY resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	14	216	3	21	0	0	0	0
6-10	210	2091	115	562	74	100	75	65
11-20	486	4560	356	1486	235	637	288	480
21-30	302	2691	234	1166	163	496	197	498
31-50	351	2433	237	1032	148	482	150	402
51-100	248	1240	155	529	86	271	79	186
101-250	62	371	51	147	40	88	26	52
251-2600	4	66	13	26	8	19	4	11
Totals	1677	13668	1164	4969	754	2093	819	1694

Table 5.8: *Global instruction scheduling for the initial architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over DHASY, and (b) DHASY resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	3	0	3	0	0	0	0
6-10	7	470	0	488	0	75	0	46
11-20	164	1239	56	1073	19	204	17	112
21-30	81	799	23	804	16	243	1	142
31-50	78	731	33	680	21	240	14	158
51-100	70	532	119	483	106	228	22	109
101-250	58	95	54	142	56	94	10	27
251-2600	9	27	11	29	11	20	3	12
Totals	461	3837	296	3702	229	1104	67	606

Table 5.9: *Global instruction scheduling for the initial architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	1	130	1	0	0	0	0	0
6-10	49	476	13	206	5	51	10	50
11-20	212	632	91	532	26	289	22	237
21-30	193	488	80	415	28	247	35	277
31-50	286	463	106	343	41	255	32	240
51-100	214	314	121	305	99	241	47	207
101-250	63	165	56	110	48	86	27	84
251-2600	16	25	7	25	3	29	4	22
Totals	1034	2693	475	1936	250	1198	177	1117

Table 5.10: *Global instruction scheduling for the initial architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	3	0	0	0	0	0	0
6-10	7	106	3	42	3	2	1	3
11-20	23	207	28	152	24	74	13	37
21-30	45	212	29	212	18	113	14	52
31-50	99	212	94	264	54	183	31	177
51-100	92	210	150	278	107	246	50	125
101-250	58	95	69	99	59	92	40	67
251-2600	11	16	12	17	8	17	2	6
Totals	335	1061	385	1064	273	727	151	467

Table 5.11: *Global instruction scheduling for the initial architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than DHASY's schedule, and (b) DHASY's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	253	0	18	0	0	0	0
6-10	116	663	68	346	19	64	20	63
11-20	192	976	220	654	141	362	144	323
21-30	185	629	177	505	106	304	112	289
31-50	179	597	235	460	174	286	131	286
51-100	154	361	148	301	106	221	103	209
101-250	68	158	84	98	88	83	40	87
251-2600	11	24	11	26	12	25	9	23
Totals	905	3661	943	2408	646	1345	559	1280

Table 5.12: *Global instruction scheduling for the initial architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than DHASY's schedule, and (b) DHASY's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	0	0	0	0	0	0	0
6-10	28	106	9	96	1	11	2	11
11-20	118	164	91	158	53	56	19	37
21-30	122	204	103	186	44	94	40	39
31-50	214	176	162	228	116	148	65	138
51-100	186	173	190	187	165	156	108	99
101-250	73	72	80	84	77	78	52	43
251-2600	10	5	16	13	16	12	6	2
Totals	751	903	651	952	472	555	292	369

Table 5.13: *Global instruction scheduling for the improved architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	1834	844	60	865	21	841	15	803	36
applu	262	171	1	162	5	154	6	154	5
apsi	1039	648	24	680	32	668	31	660	28
art	295	189	5	195	6	183	5	185	6
bzip2	480	273	6	282	6	270		266	7
crafty	2469	1149	77	1188	73	1138	20	1118	17
eon	2453	1178	42	1556	32	1525	25	1174	41
equake	190	82	1	116	3	112		108	
facerec	580	336	14	346	17	343	16	363	9
fma3d	4888	2498	77	2792	58	2272	47	2436	47
galgel	2657	1548	54	1602	70	1514	56	1440	117
gap	11518	5621	13	5706	264	5340	88	5383	104
gcc	24698	10886	729	11178	414	10800	84	10912	100
gzip	794	413	29	425	20	399	24	388	23
lucas	552	342	6	368	7	364	1	361	2
mcf	202	88	1	104	1	95	1	93	2
mesa	7003	2886	60	3611	68	3452	29	3440	101
mgrid	72	43		48		43	1	47	1
parser	2042	947	55	1018	16	959	11	956	13
perl	9219	4162	242	4364	148	4215	75	4294	69
sixtrack	4930	2818	66	2972	167	2744	116	2775	84
swim	192	164	3	163	4	156	4	149	11
twolf	4269	2000	99	2114	59	2036	19	2058	44
vortex	6613	2203	203	2538	92	2279	56	2327	39
vpr	1677	709	19	749	39	641	16	656	25
wupwise	268	133	3	129	7	118	9	125	6
Totals	91196	42331	2029	45271	1629	42661	755	42671	937

Table 5.14: *Global instruction scheduling for the improved architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the optimal scheduler failed to complete within a 10-minute time limit, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
ammp	1950	706	28	905	19	851	12	843	30
applu	266	154		189	4	170	4	165	3
apsi	1281	650	6	916	15	855	15	856	16
art	448	178	2	275	5	227	4	240	6
bzip2	495	245	2	303	3	288		286	1
crafty	2630	942	25	1315	16	1172	5	1172	10
eon	3059	783	26	1334	18	1237	12	1221	30
equake	195	78	1	123		106		112	1
facerec	643	348	7	444	12	403	12	404	2
fma3d	6009	2338	58	3144	44	2783	33	2875	59
galgel	3046	1450	36	2075	35	1900	54	1832	116
gap	11524	4938	48	5901	61	5485	25	5549	65
gcc	25053	9615	440	11845	168	10971	49	11141	87
gzip	810	340	8	452	10	408	20	410	21
lucas	545	343	9	399	3	384	3	381	3
mcf	236	79	1	122	1	109		111	1
mesa	7775	2900	45	4377	34	4010	24	4289	81
mgrid	78	46		56	1	49	2	49	2
parser	2050	847	48	1039	19	978	12	990	14
perl	9387	3501	154	4447	57	4184	49	4509	58
sixtrack	5505	2716	53	3627	108	3140	106	3221	80
swim	190	149	7	163	5	156	5	142	20
twolf	4330	1815	81	2349	28	2067	12	2132	29
vortex	6601	2449	105	3388	85	2974	52	3073	46
vpr	1752	612	13	886	11	749	3	776	13
wupwise	280	123	3	158	3	139	6	141	7
Totals	96138	38345	1206	50232	765	45795	519	46920	801

Table 5.15: *Global instruction scheduling for the improved architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of superblocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	15764	9304	59.0	8388	59.6	9406	59.7	9437	59.9
6-10	22720	10688	47.0	11815	52.0	11502	50.6	11424	50.3
11-20	27596	11585	42.0	12968	47.0	11745	42.6	12149	44.0
21-30	10985	4304	39.2	4380	39.9	4054	36.9	3941	35.9
31-50	8409	3494	41.6	3546	42.2	3184	37.9	2982	35.5
51-100	4287	2116	49.4	2254	52.6	1917	44.7	1867	43.6
101-250	1274	747	58.6	800	62.8	744	58.4	759	59.6
251-2600	161	93	58.5	109	68.6	109	68.6	112	70.4
Totals	91196	42331	46.4	45271	49.6	42661	46.8	42671	46.8

Table 5.16: *Global instruction scheduling for the improved architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found an improved schedule over the best heuristic schedule, and (b) the percentage of superblocks with improved schedules, for various architectures.

		1-Issue		ppc603e		ppc604		IA-64	
	# blocks	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	16274	9066	55.7	9466	58.2	9416	57.9	9434	58.0
6-10	23726	9173	38.7	12093	51.0	11224	47.3	11829	50.0
11-20	28713	10135	35.3	14566	50.7	12369	43.1	13117	45.7
21-30	11566	3723	32.2	5175	44.7	4484	38.8	4443	38.4
31-50	9123	3224	35.3	4498	49.3	4023	44.1	3866	42.4
51-100	5137	2132	41.5	3199	62.3	3085	60.1	3011	58.6
101-250	1401	760	54.2	1075	76.7	1043	74.4	1093	78.0
251-2600	198	132	66.7	160	80.8	151	76.3	127	64.1
Totals	96138	38345	39.9	50232	52.3	45795	47.6	46920	48.8

Table 5.17: *Global instruction scheduling for the improved architectural model before register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the superblocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	10.5	31.0	14.6	59.2	14.6	59.2	15.1	59.2
6-10	5.3	31.0	7.3	50.0	7.3	50.0	7.9	58.3
11-20	3.5	78.1	4.6	46.5	4.3	47.2	5.2	44.0
21-30	2.7	37.9	2.9	31.5	2.7	30.8	2.9	42.2
31-50	2.4	44.6	2.9	43.4	2.7	43.4	2.9	51.3
51-100	2.4	82.3	3.0	38.3	2.8	39.6	3.3	56.6
101-250	2.4	38.1	2.8	36.8	2.9	43.4	3.2	36.8
251-2600	3.7	51.6	4.5	54.2	2.9	28.0	4.2	39.8
Totals	5.3	82.3	7.0	59.2	7.0	59.2	7.6	59.2

Table 5.18: *Global instruction scheduling for the improved architectural model after register allocation.* Average and maximum percentage improvements in schedule length of optimal schedule over the best heuristic schedule, for various architectures. The average is over *only* the superblocks in the SPEC 2000 benchmark suite for which the optimal scheduler found an improved schedule.

	1-Issue		ppc603e		ppc604		IA-64	
	avg.	max.	avg.	max.	avg.	max.	avg.	max.
3-5	10.6	33.1	15.0	59.2	14.9	59.2	15.4	59.2
6-10	5.0	45.1	8.2	50.0	7.9	50.0	8.8	58.3
11-20	3.2	50.6	5.4	49.3	5.0	41.9	6.0	56.4
21-30	2.1	58.1	3.4	38.9	3.1	32.5	3.8	49.4
31-50	1.9	34.5	3.5	51.0	3.6	41.9	4.2	48.4
51-100	2.1	47.5	3.7	39.7	3.8	37.5	5.0	56.6
101-250	1.7	64.1	3.2	40.0	3.2	25.5	4.7	51.8
251-2600	2.1	18.6	2.4	40.3	2.4	17.3	4.2	18.5
Totals	5.1	64.1	7.3	59.2	7.3	59.2	8.1	59.2

Table 5.19: *Global instruction scheduling for the improved architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over DHASY, and (b) DHASY resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	19	125	4	76	3	3	3	3
6-10	224	2148	159	668	47	159	62	96
11-20	446	4627	357	2061	215	942	268	532
21-30	210	2528	175	1212	153	580	190	553
31-50	242	2316	190	1289	129	750	171	522
51-100	183	1221	179	755	107	441	129	295
101-250	44	373	71	245	51	165	90	104
251-2600	5	62	6	51	10	35	6	26
Totals	1373	13400	1141	6357	715	3075	919	2131

Table 5.20: *Global instruction scheduling for the improved architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) critical path resulted in an improved schedule over DHASY, and (b) DHASY resulted in an improved schedule over critical path, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	7	0	7	0	1	0	1
6-10	8	476	22	548	3	104	3	80
11-20	169	1252	185	1280	77	362	48	272
21-30	95	803	107	986	50	438	55	269
31-50	84	702	182	942	135	563	91	432
51-100	72	547	212	739	179	491	112	412
101-250	55	139	124	224	134	176	85	121
251-2600	9	29	39	37	39	39	27	17
Totals	492	3955	871	4763	617	2174	421	1604

Table 5.21: *Global instruction scheduling for the improved architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	1	49	5	12	4	1	4	1
6-10	59	566	30	377	42	132	42	161
11-20	437	1048	261	703	229	506	124	428
21-30	377	480	255	576	140	414	166	447
31-50	492	599	405	608	291	523	255	407
51-100	441	476	341	409	250	336	222	375
101-250	168	157	114	120	104	87	97	123
251-2600	14	14	20	18	17	18	15	17
Totals	1989	3389	1431	2823	1077	2017	925	1959

Table 5.22: *Global instruction scheduling for the improved architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the optimal scheduler found a schedule with lower register pressure than both heuristic schedules, and (b) a heuristic schedule had lower register pressure than the optimal schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	23	1	3	1	0	1	0
6-10	28	234	62	159	46	142	47	123
11-20	285	673	279	482	159	363	152	296
21-30	217	297	221	355	189	347	215	293
31-50	330	409	338	468	300	544	295	472
51-100	378	268	335	406	284	428	271	397
101-250	136	87	114	134	103	119	104	97
251-2600	16	14	6	10	9	6	5	8
Totals	1390	2005	1356	2017	1091	1949	1090	1686

Table 5.23: *Global instruction scheduling for the improved architectural model before register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than DHASY's schedule, and (b) DHASY's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	7	2	68	0	0	0	0
6-10	117	518	102	377	35	85	36	77
11-20	194	916	226	744	148	360	140	320
21-30	115	540	173	454	103	288	98	285
31-50	216	485	253	484	221	280	137	279
51-100	178	307	157	314	109	239	103	235
101-250	63	141	55	121	66	84	41	90
251-2600	6	17	8	28	9	17	5	19
Totals	889	2931	976	2590	691	1353	560	1305

Table 5.24: *Global instruction scheduling for the improved architectural model after register allocation.* Number of superblocks in the SPEC 2000 benchmark suite with more than two instructions where (a) the critical path schedule had lower register pressure than DHASY's schedule, and (b) DHASY's schedule had lower register pressure than the critical path schedule, for various architectures.

	1-Issue		ppc603e		ppc604		IA-64	
	(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)
3-5	0	1	0	1	0	1	0	1
6-10	28	113	40	117	9	27	14	19
11-20	119	164	134	183	78	76	63	57
21-30	129	200	144	205	101	99	89	94
31-50	226	184	265	241	221	126	127	102
51-100	188	166	212	167	179	153	142	109
101-250	67	74	90	81	79	70	65	55
251-2600	5	6	12	6	12	10	8	4
Totals	762	908	897	1001	679	562	508	441

5.4 Summary of Results

Schedules produced by a list scheduler are sufficiently near optimality for global instruction scheduling when scheduling for an idealized architectural model. While list scheduling produces better results for local instruction scheduling, the results presented in this chapter suggest that the cost of an optimal scheduler for global instruction scheduling still heavily outweighs the improvements in schedule quality that could be obtained by using an optimal scheduler, provided that an idealized architectural model is used. When a realistic architectural model is used, list scheduling performs poorly for all architectures, and the optimal scheduler is able to make significant improvements to schedule cost. This work demonstrates that list scheduling is not near-optimal in all cases and provides motivation for further work in superblock scheduling.

Chapter 6

Conclusions and Further Work

This thesis examined the optimality of the list scheduling algorithm for both simplistic and realistic architectural models. Experiments were done for both local and global instruction scheduling.

For the local instruction scheduling problem, the list scheduler was invoked using both a critical path heuristic and one created by Shieh and Papachristou. When scheduling for the idealized architectural model, the list scheduler solved 98.6%-99.9% of the basic blocks in the benchmark suite optimally. For the improved architectural model, the list scheduler produced optimal schedules for 94.2%-97.8% of the basic blocks. I also found that Shieh and Papachristou's heuristic performed slightly better than the critical path heuristic, and that there was negligible differences between the two heuristics and the optimal scheduler with respect to register pressure.

For the global instruction scheduling problem, the critical path heuristic and the DHASY heuristic were used for comparison against the optimal scheduler. When scheduling for the idealized architectural model, the list scheduler solved 91.2%-97.5% of superblocks optimally. However, the list scheduler was only optimal for 47.7%-60.1% of superblocks when scheduling for a realistic architectural model, and schedules produced by the optimal schedule were an improvement of 5.3%-8.1% on average over non-optimal schedules produced by the list scheduler. As expected, DHASY yields better schedules than the critical path heuristic. As with local instruction scheduling, there is little difference between heuristic schedules and optimal schedules with respect to register pressure.

The most significant conclusion of this thesis is that list scheduling is sufficiently close to optimality in practice for local instruction scheduling but not for global instruction scheduling. There is almost no need for optimal schedulers of any kind when scheduling basic blocks, as the cost of invoking an optimal scheduler will generally outweigh the cost of list scheduling, and there will only be benefits for a small number of blocks which may not even be significant to the execution time of a particular application. This is not the case for global instruction scheduling, and other superblock scheduling algorithms must be investigated in order to produce lower-cost schedules for superblocks being scheduled on realistic architectures.

In order to take advantage of an optimal scheduler, one possibility might be to invoke an optimal scheduler on each block with a short time limit. If the limit was exceeded, the schedule produced by the list scheduling algorithm as an upper bound for the optimal schedule could be returned. This would likely improve schedule cost overall without incurring too dramatic a penalty to compilation time.

Another result of this thesis is evidence that secondary features should be examined more closely for local instruction scheduling heuristics. Shieh and Papachristou's heuristic, differing only in secondary features from the critical path heuristic, achieved better overall performance. A comprehensive study with a large number of features on a realistic architectural model might result in better schedules being produced.

Finally, there seemed to be minimal differences in register pressure between the optimal scheduler and any heuristic schedule. There was also no one scheduler or heuristic that strongly outperformed the others with respect to register pressure. While it might be desirable to construct a scheduler or heuristic that accounted for register pressure, it appears that among those that do not, there is no best choice to minimize register pressure, and the focus can be solely on minimizing schedule cost.

6.1 Further Work

There are two relevant architectural features which were not modelled as part of this thesis. I did not model instructions that can be issued on more than one type of functional unit, nor did I handle the case when not all resources, including issue slots, functional units, and registers, were available to the scheduler when scheduling each new block. Both of these are properties that could be included in a more complete study of architectural properties if suitable test data were available.

Another avenue of research would be to design an optimal scheduler that accounted for register pressure. Our optimal scheduler ignores register pressure entirely. One possible way of implementing this would be to invoke the scheduler with a given number of available registers R. The scheduler would then either find an optimal cost schedule using at most R registers or assert that no such schedule exists.

Bibliography

[1] P. Baptiste and C. Le Pape. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 600–606, Montréal, 1995.

[2] D. Bernstein and I. Gertner. Scheduling expressions on a pipelined processor with a maximal delay of one cycle. *ACM Transactions on Programming Languages and Systems*, 11(1):57–66, 1989.

[3] R. J. Blainey. Instruction scheduling in the TOBEY compiler. *IBM J. Res. Develop.*, 38(5):577–593, 1994.

[4] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. van Hensbergen, and L. Zhang. Mambo - a full system simulator for the PowerPC architecture.

[5] R. Bringmann. Enhancing instruction level parallelism through compiler-controlled speculation. Technical Report 1897, Urbana, Illinois, 1994.

[6] C. Chekuri, R. Johnson, R. Motwani, B. Natarajan, B. R. Rau, and M. Schlansker. Profile-driven instruction level parallel scheduling with application to superblocks. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-29)*, pages 58–67, Paris, 1996.

[7] R. Debruyne and C. Bessière. Domain filtering consistencies. *J. Artificial Intelligence Research*, 14:205–230, 2001.

[8] B. L. Deitrich and W. W. Hwu. Speculative hedge: Regulating compile-time speculation against profile variations. In *Proceedings of the 29th Annual*

IEEE/ACM International Symposium on Microarchitecture (Micro-29), pages 70–79, Paris, 1996.

[9] A. E. Eichenberger and W. M. Meleis. Balance scheduling: Weighting branch tradeoffs in superblocks. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-32)*, pages 272–283, Haifa, Israel, 1999.

[10] M. A. Ertl and A. Krall. Optimal instruction scheduling using constraint logic programming. In *Proceedings of 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 75–86, Passau, Germany, 1991.

[11] P. Faraboschi, J. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, 89(11):1638–1659, 2001.

[12] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Computers*, 30(7):478–490, 1981.

[13] J. Fisher. Global code generation for instruction-level parallelism: Trace scheduling-2. Technical Report HPL-93-43, Hewlett-Packard Laboratories, 1993.

[14] P. B Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 11–16, New York, NY, USA, 1986. ACM Press.

[15] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):263–269, 1969.

[16] M. Heffernan and K. Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8:427–451, 2005.

[17] J. Hennessy and T. Gross. Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, 1983.

[18] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[19] S. Hoxey, F. Karim, B. Hay, and H. Warren. *The PowerPC Compiler Writer's Guide.* Warthman Associates, 1996.

[20] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, 1993.

[21] IBM. *PowerPC 604e RISC Microprocessor User's Manual.* 1998.

[22] Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture.* 2002.

[23] Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 3: Instruction Set Reference.* 2002.

[24] Intel. *IA-32 Intel Architecture Software Developer's Manual, Volume 3A: System Programming Guide, Part 1.* 2006.

[25] D. Kästner and S. Winkel. ILP-based instruction scheduling for IA-64. In *Proceedings of the SIGPLAN 2001 Workshop on Languages Compilers, and Tools for Embedded Systems (LCTES)*, pages 145–154, Snowbird, Utah, 2001.

[26] J. Liu and F. Chow. A near-optimal instruction scheduler for a tightly constrained, variable instruction set embedded processor. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 9–18, Grenoble, France, 2002.

[27] A. M. Malik, J. McInnes, and P. van Beek. Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. Technical Report CS-2005-19, School of Computer Science, University of Waterloo, 2005.

[28] A. M. Malik, T. Russell, M. Chase, and P. van Beek. Optimal superblock instruction scheduling for multiple-issue processors using constraint programming. Technical report, School of Computer Science, University of Waterloo, 2006.

[29] S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[30] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

[31] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[32] T. Russell. Personal communication, August, 2006.

[33] T. Russell, A. M. Malik, M. Chase, and P. van Beek. Learning basic block scheduling heuristics from optimal data. In *Proceedings of the 15th CASCON*, Toronto, 2005.

[34] P. J. Schielke. *Stochastic Instruction Scheduling*. PhD thesis, Rice University, 2000.

[35] J.-J. Shieh and C. Papachristou. On reordering instruction streams for pipelined computers. *SIGMICRO Newsl.*, 20(3):199–206, 1989.

[36] G. Shobaki and K. Wilken. Optimal superblock scheduling using enumeration. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-37)*, pages 283–293, Portland, Oregon, 2004.

[37] M. Smotherman, S. Krishnamurthy, P. S. Aravind, and D. Hunnicutt. Efficient DAG construction and heuristic calculation for instruction scheduling. In *Proceedings of the 24th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-24)*, pages 93–102, Albuquerque, New Mexico, 1991.

[38] P. van Beek and K. Wilken. Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 625–639, Paphos, Cyprus, 2001.

[39] K. Wilken, J. Liu, and M. Heffernan. Optimal instruction scheduling using integer programming. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, Vancouver, 2000.