WAKEUP-SET SCHEDULING FOR LARGE INSTRUCTION WINDOW PROCESSORS

BY

KEVIN MICHAEL WOLEY

B.S., University of Illinois at Urbana-Champaign, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Out-of-order processor performance is limited by instruction scheduler size. Current "issue buffer" based instruction scheduler implementations do not scale with the size of the instruction window. We propose distributing the functions of instruction wakeup and selection-issue across separate structures specialized for each task and conclude that (1) limiting the "bandwidth of wakeup" to the width of the pipeline does not significantly impact performance, (2) the number of instructions awaiting wakeup by a particular physical tag does not significantly increase with the scheduler size, and (3) the number of instructions which are ready for issue is considerably smaller than the size of the scheduler itself.

Thus we propose the wakeup-set scheduler. Instructions with unsatisfied dependencies are placed in a bankable, limited sized wakeup-sets array where they await the production of their operands. A separate ready-instruction queue holds the instructions which are ready to issue for execution. As register values are produced, they index their wakeup-sets to discover waiting instructions. When an instruction's dependencies are satisfied, it moves from its wakeup-set to the ready-instruction queue from where it may be issued for execution. We present an implementation of a wakeup-set scheduler and evaluate its performance. Our results indicate that the wakeup-set scheduler performs within 5% of a highly idealized, 1024 entry instruction scheduler, and within 2% of a 64 entry scheduler in an aggressive superscalar.

For Patricia Ann and Michael Patrick,
without whom I have nothing,
and to whom I owe everything.

ACKNOWLEDGMENTS

Mathew Ian Frank has provided, on levels much higher than simple academic advisory council, insights and guidance untold. To him I give my gratitude; I look forward to continuing my journey of higher education with him as my adviser and confidant.

Sanjay Patel has supported my graduate career by providing guidance and insight into my work without which I would have never started down the path I am on.

Additionally, I would like to thank the members of the Implicitly Parallel Architectures group for their insights, development efforts, and comradery which were the underpinnings of this work. I would specifically like to thank Kshitiz Malik, Kevin Stephano, Sam Stone, and Todd Rafacz for their simulator development efforts upon which this work was built.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

High frequency, out-of-order processors often have a critical path which is at the heart of the dynamic instruction wakeup and select algorithm [1]. This critical path is a serious impediment to both increasing clock frequency and increasing the size of the scheduler. In many designs it is the case that instructions which are ready to be issued for execution are intermingled with the set of instructions which are not ready to be issued in the same structure. Using one structure for the two distinct purposes of wakeup and selection-issue exacerbates the sensitivity to this critical path. However, using one structure for both sets of instructions is not strictly required; if the functions of wakeup and selection-issue are implemented by separate algorithms and structures the wakeup logic can be simplified and the search space for instruction selection-issue greatly reduced.

We show that, as the size of the instruction window increases, the number of dynamic instructions which are ready to issue grows at a rate substantially lower than that of the number of instructions which have outstanding dependencies. Therefore, the process of scheduling the set of instructions which have had their data dependencies satisfied across the available functional units is a more tractable problem than managing the wakeup of an increasingly larger set of instructions. Thus we focus on building a mechanism which enables efficient wakeup of very large pools of instructions while simplifying the selection and issue logic by removing the need to tease out those instructions which are available for issue from those which are not.

In this thesis we study the *wakeup-set scheduler*. A *wakeup-set* is the set of instructions which await wakeup on a single tag. A *tag* is a resource identifier on which an instruction can be dependent, such as a physical register. An instruction is a member of as many wakeup-sets as it has unsatisfied dependencies. In conventional issue buffer implementations of out-of-order schedulers the wakeup-sets are discovered dynamically at the time the scheduler is informed that resource associated with a tag is available. At a minimum this requires an associative search of all instructions which are known to be waiting on tags. More commonly an associative search through all instructions, ready and waiting, is performed each time a tag is transmitted to the scheduler. This just-in-time computation of wakeup-sets via an associative search limits the ability of the scheduler to scale effectively.

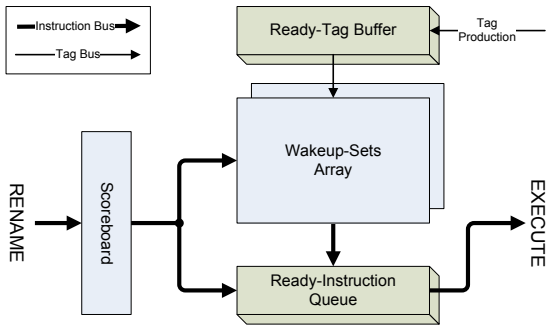


Figure 1.1 Wakeup-Set Scheduler

Figure 1.1 depicts the wakeup-set scheduler. In a wakeup-set scheduler instructions which do not yet have their data dependencies satisfied at dispatch are placed in wakeup-sets to await their dependencies. This effectively precomputes the wakeup-set for each tag as dependent instructions are dispatched. Organizing dependent instructions into wakeup-sets limits the search space at tag production to only those instructions which can possibly be awakened by the tag produced. Instructions which have their data dependencies satisfied when they enter the scheduling stage go directly to a *ready-instruction queue* where they await selection and issue. As instructions wake up from the *wakeup-sets array* they are moved from their wakeup-set(s) to the ready-instruction queue. This approach allows the scheduling window to scale with the combined size of the wakeup-sets array and ready-instruction queue while enabling both structures to be tailored to meet their algorithmic needs.

The remainder of the thesis is organized as follows. In Section 2, we present an idealized out-of-order instruction scheduler and observe several factors which motivate the wakeup-set scheduler. In Section 3, the performance implications of reducing the wakeup bandwidth are discussed and the general design of the wakeup-set scheduler is given. In Section 4, an implementation of a wakeup-set structured scheduler is discussed and its performance evaluated. In Section 5, we discuss and prior art related to our approach. Section 6 concludes.

CHAPTER 2
MOTIVATION

In this chapter we empirically observe the wakeup bandwidth of an ideal scheduler and the dynamic distribution of ready to waiting instructions which are held in the scheduler. We also discuss instruction distributions in terms of wakeup-sets. Section 2.1 describes our simulation environment and the machine configurations used throughout this thesis. In Section 2.2 we give a working definition of an "ideal" out-of-order instruction scheduler. The remaining sections motivate the design of a wakeup-set scheduler. Section 2.3 discusses the distribution of instructions in the scheduler as the scheduler size increases. Section 2.4 demonstrates that the bandwidth of wakeup is close to the machine width on average. Section 2.5 characterizes scheduled instructions as members of wakeup-sets as scheduler size is varied.

2.1 Simulation Environment

All simulations are performed on an execution-driven simulator of a 64-bit MIPS pipeline. The simulator executes all instructions, including those on mispredicted paths, and the results produced by retiring instructions are validated against a trace generated by an architectural simulator. Our processor performs Alpha-21264 style renaming which enables checkpoint recovery. The back-end of our processor is equipped with identical, fully pipelined functional units to match the machine's width. We model a memory subsystem which can handle an infinite number of outstanding memory requests. To simplify the discussion of the scheduler, a store-set predictor is not implemented.

We simulate two different processors, a baseline superscalar that supports up to 128 instructions in flight simultaneously and an aggressive superscalar that supports up to 1024 instructions in flight simultaneously. The aggressive simulator has an enhanced branch predictor and is able to fetch past several branches in a cycle to support its larger instruction window. The machine configuration parameters used in this thesis, unless otherwise noted, are given in Table 2.1.

Table 2.1 Simulation Parameters

Parameter	Baseline	Aggressive
Pipeline Width	4 instructions/cycle	8 instructions/cycle
Fetch Bandwidth	1 branch/cycle	8 branches/cycle
Renamer	128 checkpoints	1024 checkpoints
Scheduling Window	128 entries	1024 entries
Physical Registers	128	1024
Load / Store Queues	128 entries each	1024 entries each
Reorder Buffer	128 entries	1024 entries
Function Units	4 identical, fully pipelined	8 identical, fully pipelined
Branch Predictor	8 Kbit Gshare	+ 80% mispredicts turned to correct predictions by an "oracle"
Misprediction Penalty	8 cycles	
Branch Target Buffer	4096 entries, 4-way set associative	
Split L1 Caches	*Instruction:* 8 KB, 2-way set associative, 128 B lines, 1 cycle latency *Data:* 8 KB, 4-way set associative, 64 B lines, 1 cycle latency	
Unified L2 Cache	512 KB, 8-way set associative, 128 B lines, 10 cycle latency	
Main Memory	100 cycle latency	

We simulated 19 of the 26 SPEC 2000 benchmarks,[1] each with the lgred or mdred Minnesota Reduced inputs. Benchmarks were run to completion or to 300 000 000 instructions.

2.2 Idealized Out-of-Order Instruction Scheduling

As instructions finish renaming they are dispatched to the scheduler, which is responsible for buffering instructions until such time as they can be issued to functional units. The scheduler performs two essential tasks: instruction wakeup and selection-issue. Instructions which enter the scheduler may or may not have all of their dependencies satisfied such that they are ready to issue. Instructions which are not ready to issue have their dependencies tracked and are "awakened" by the producers of their tags. A tag is an identifier which is associated with a physical or virtual resource, such as a physical register or store-set. We refer to the transmitting of a tag to the scheduler to enable wakeup of dependent instructions as *tag production*. Instructions which have all of their dependencies satisfied are available for selection. Selection is the process of choosing which instructions will be allowed to move on to execution. The structure

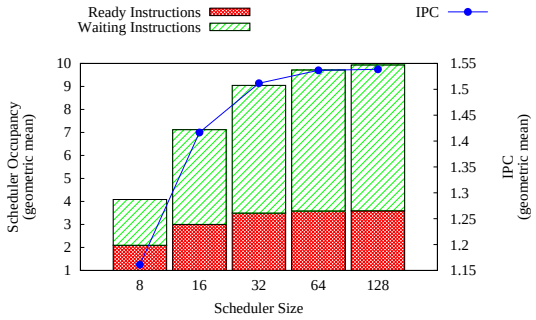
[1] We lack compiler support for the Fortran 90 benchmarks,(`galgel`, `facerec`, `lucas` and `fma3d`) and runtime library support for `wupwise` `sixtrack` and `eon`.

which holds all of the instructions in the scheduler is commonly called the *issue buffer*; we will use this term in reference to the conventional out-of-order scheduling structure.

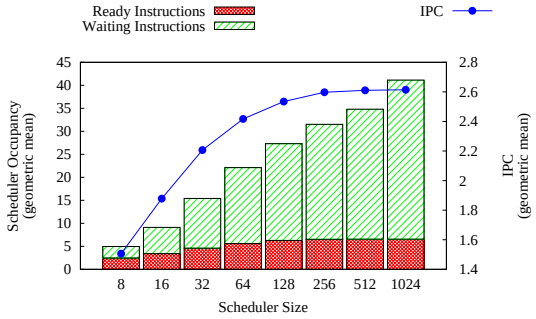
The out-of-order instruction scheduler may be idealized along several dimensions; for our purposes the "ideal" scheduler is as follows. Our ideal scheduler is able to awaken an unlimited number of instructions to be issued within a single cycle. Furthermore, every instruction which is awakened within a cycle is also available for issue within that same cycle. This scheduler is only constrained by the total number of instructions it can contain, the number of instructions which can be inserted from rename, and the number of instructions which are allowed to be issued within a single cycle. These constraints are imposed to represent the implementation details which are assumed to be immutable in any physical out-of-order scheduler design. The size of the scheduler for the 4-wide base and 8-wide aggressive configurations are listed in Figure 2.1 unless otherwise indicated. In both configurations the number of instructions per cycle which are dispatched, and those which are issued, are bounded by their respective superscalar machine widths.

Ideally all instructions dependent on a data value, barring those with additional dependencies, are awakened and issued the cycle *prior* to that value's production. Issuing an instruction a cycle prior to its data value being produced allows it to obtain operand values from register bypass logic. This avoids the need to wait for register writeback and reduces a significant critical path. In hardware scheduler implementations this sort of aggressive wakeup and selection is typically performed for instructions for which static dependencies are known. However, instructions like memory operations have a variable delay complicating hardware implementable wakeup and selection. In our idealized scheduler we issue from the set of ready instructions ideally for all dependencies, meaning that we issue as early as possible using perfect knowledge about dynamic latencies.

Since the scheduler is constrained by the number of instructions it can issue per cycle, having more instructions ready to issue than issue slots means a choice must be made between ready instructions. Under the most ideal circumstances the instructions most critical to performance should be given issue priority. Without the benefit of criticality analysis [2] one common scheduling policy is to issue the oldest ready instruction first. In our idealized scheduler we implement the oldest-first issue policy to select from among the set of ready instructions to issue.



(a) Base Configuration



(b) Aggressive Configuration

Figure 2.1 Scheduler Occupancy

The number of ready instructions in the scheduler does not increase with the total occupancy of the scheduler as the scheduler size increases. There is a stronger correlation between IPC and the mean ready instructions than between IPC and total scheduler size or occupancy.

2.3 Scheduler Occupancy

2.3.1 Scheduler size

We first observe the occupancy and distribution of ready instructions as a function of scheduler size. In Figure 2.1 we observe that increasing the scheduler size increases the mean occupancy of the scheduler as expected. The primary observation is that *the number of ready to issue instructions in the scheduler increases at lower rate than the total occupancy of the scheduler.* The number of ready to issue instructions becomes relatively constant beyond 64 scheduler entries in the base configuration and 256 in the aggressive. Note that we include instructions which have already issued but have not yet completed execution in our definition of "ready to issue" since it is common that these instructions must remain in the scheduler until they complete execution.

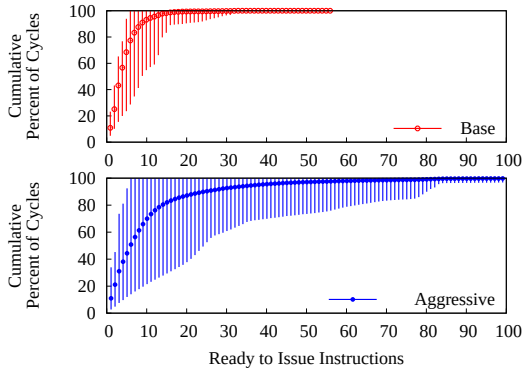


Figure 2.2 Ready to Issue Instructions

In both the base and aggressive configurations the total number of ready to issue instructions is considerably smaller than the total scheduler size. The base configuration scheduler is occupied by (12 mean, 29 max) or fewer ready instructions for 95% of execution cycles. In the aggressive configuration, (38 mean, 83 max) instructions or fewer are ready to issue for 95% of execution cycles.

In the base configuration, increasing the scheduler size from 8 to 16 increases the geometric mean number of ready to issue instructions from 2.09 to 3.00, whereas increasing the scheduler size from 32 to 128 yields a modest increase from 3.49 to 3.59. IPC appears to be strongly correlated with the number of instructions which are are ready to issue. This is seen as the scheduler size increases from 8 to 32, yielding an IPC increase of 0.35 compared to the modest IPC increase of 0.03 between 32 and 128 scheduler entries.

Under the aggressive configuration, increasing the scheduler size from 256 to 1024 only increases the mean number of ready to issue instructions from 6.52 to 6.55. This is accompanied by a modest increase of in IPC of 0.017. However, increasing the scheduler size from 8 to 32 increases the average number of ready instructions from 2.46 to 4.59, with an IPC increase of 0.70. As the scheduler size is increased from 32 to 128 the average number of ready instructions goes from 4.59 to 6.28, with an IPC increase of 0.33. These results indicate that there is a strong correlation between the number of ready instructions and the mean IPC achieved.

2.3.2 Ready instruction cumulative distribution

Figure 2.2 shows the cumulative distributions of the number of instructions which are ready to issue in the scheduler under the base configuration with 128 scheduler entries and the aggressive configuration with 1024. As discussed above, the geometric mean number of instructions which are ready to issue is considerably smaller than the size of the scheduler itself.

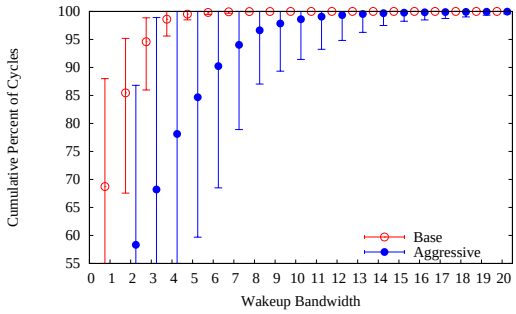


Figure 2.3 Observed Wakeup Bandwidth

The min, max, and geometric mean of the average wakeup bandwidth across all benchmarks is shown as a cumulative distribution. We note that the number of instructions which are awakened per cycle is less than the superscalar width on average. The 4-wide base configuration awakens 4 or fewer instructions in over 95% of its machine cycles, whereas the 8-wide aggressive configuration awakens 8 or fewer instructions in over 95% of its machine cycles.

Figure 2.2 shows that over 95% of execution cycles have fewer than 12 ready instructions in the scheduler, which is more than three times the geometric mean. For the aggressive configuration, over 95% of the execution cycles have 38 or fewer ready instructions, which is nearly six times the geometric mean. We will use these facts to motivate the creation of a structure in the wakeup-set scheduler to hold only those instructions which are ready to issue separate from those which are waiting to awaken.

2.4 Wakeup Bandwidth

Another observation we make on the idealized scheduler is the *bandwidth of wakeup*, which is the rate at which instructions transition from waiting on operands to being ready to issue. We observe that for over 95% of cycles, *the machine width or fewer instructions are awakened by the scheduler per cycle.* This is shown in Figure 2.3, which shows the observed wakeup bandwidth for the base configuration with 128 scheduler entries and the aggressive configuration with 1024 scheduler entries.

Figure 2.3 is representative of the trend across all scheduler sizes simulated. The number of instructions awakened per cycle, discounting cycles which awaken no instruction, increases slowly as the scheduler size increases. The mean number of instructions awakened per cycle increases from 1.36 to 1.89 for the base configuration as the scheduler size is varied from 8 to 128. As the aggressive configuration scheduler size is varied from 8 to 1024, the mean number of instructions awakened per cycle goes from 1.60 to 3.60.

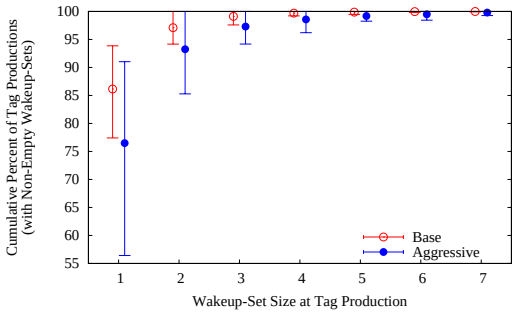


Figure 2.4 Wakeup-Set Size Cumulative Distribution

The min, max, and geometric mean of the average wakeup-set size across all benchmarks is shown as a cumulative distribution. The results shown indicate that the size of the wakeup-sets is four or less in over 98.5% of the tag production events which have dependent instructions. Note that the total percentage shown does not include all tag production events; shown are only the tag production events which are to wakeup-sets with 1 or more entries. The overall percent of tag production events which have empty wakeup-sets is 8.55% (max 41.55%) and 4.26% (max 21.62%) for the base and aggressive configurations, respectively.

The maximum number of instructions which are able to awaken per cycle is potentially the most important design criterion when considering separating the functions of wakeup and selection-issue. Knowing that the common case wakeup bandwidth is reasonably bounded enables the exploration of scheduler designs which impose a hard limit on the wakeup bandwidth. The design of the wakeup-set scheduler relies heavily on this fact, which is explored further in Section 3.

2.5 Wakeup-Sets Size

If we consider the instructions awaiting wakeup as members of wakeup-sets, as discussed in Section 1, we observe two properties about the ideal instruction scheduler. First, *the size of the wakeup-sets associated with a given tag do not increase with the scheduler size.* The geometric mean wakeup-set size at tag production under the base configuration ranges from 1.10 to 1.17 as the scheduler size increases from 8 to 128. With the aggressive configuration the mean wakeup-set size increases from 1.15 to 1.35 as the scheduler size goes from 8 to 1024.

The corollary is that the *number of wakeup-sets increases as the instruction scheduler size increases*. The geometric mean of the number of wakeup-sets as the base configuration scheduler size varies from 8 to 128 entries goes from 2.71 to 8.49 wakeup-sets. The geometric mean

of wakeup-sets under the aggressive configuration varies from 2.98 to 44.55 as the scheduler size increases from 8 to 1024 entries.

We believe this to be an inherent property of wakeup-sets; the size of a wakeup-set is a function of the throughput from dispatch to issue more than a function of scheduler size. The size of a wakeup-set is limited by the rate at which instructions enter the scheduler and the rate at which instructions complete. The average size of a wakeup-set is then a function of the slack in the rates of dispatch and the production of instruction dependencies. Thus an in-flight instruction has the potential to build a wakeup-set only as large as the number of dependent instructions dispatched before the instruction completes execution.

Another key observation is that *the majority of tag productions have wakeup-sets smaller than five instructions.* This is shown in Figure 2.4, which shows results from the base and aggressive configurations with 128 and 1024 scheduler entries, respectively. These cumulative distributions show that a very few tags sent to the scheduler have more than four instructions waiting on them. Figure 2.4 shows that in both the base and aggressive configurations fewer than 1.5% of all produced physical tags have more than four instructions waiting on them. This fact is used in Section 3.2 to design a wakeup-set structured scheduler which has limited sized wakeup-sets.

2.6 Motivation Summary

Knowing the distribution of the instructions inside the idealized scheduler as well as how wakeup-sets sizes scale with scheduler size provides insight as to how the separation of wakeup from selection will effect performance. In particular, it motivates a scheduler design which at minimum must support a wakeup bandwidth that equals the superscalar width to achieve optimal performance. The results shown also support the insight that the engineering effort in scaling the scheduler to larger sizes should be focused on supporting the wakeup of an increasingly larger number of instructions.

There is naturally a diminishing return to increasing the total scheduler size as shown by these and other results [3]. This is largely due to the increase in scheduler size not directly translating into a larger number of ready instructions. However, to increase the average number of ready instructions, the scheduler must be able to buffer a large number of waiting instructions in a scalable, efficient manner. Understanding how the scheduler scales will enable us to reach the scheduler sizes required to extract the upper-bound of the performance limit which has not been previously achieved.

CHAPTER 3
DESIGN

A wakeup-set structured scheduler introduces several design constraints which are not present in conventional "issue buffer" scheduler implementations. The primary implication of having separate wakeup and selection-issue structures is the wakeup bandwidth, the number of instructions which are allowed to awaken per cycle. In Section 3.1 we show that performance is not significantly degradated over the ideal by limiting the wakeup bandwidth to the width of the pipelined machine. We then propose a high-level scheduler design in Section 3.2 which achieves complexity effectiveness via wakeup-set scheduling.

3.1 Wakeup Bandwidth Limited Instruction Scheduling

In this section we study the effect of restricting the number of instructions which can be awakened per cycle. The purpose of this study is to determine to what degree wakeup bandwidth effects performance. We will demonstrate that limiting the wakeup bandwidth to at least the width of the machine does not appreciably effect performance.

When an absolute wakeup limit is enforced, a policy to determine which instructions to awaken in a cycle is required. The production of a resource tag enables a set of instructions to be awakened associated with that tag. If a limited number of instructions can be awakened within a cycle, and more than one tag may arrive per cycle, it may not be possible to awaken all of the instructions for which tags are available.

Assuming that instructions are gathered into wakeup-sets associated with the tag that may awaken them, there are at least two options for choosing among the available wakeup-sets. The first is to attempt to awaken an entire wakeup-set before moving onto the next; we call this "set-first wakeup." The other is to attempt to awaken only one instruction per set; we call this "set-distributed wakeup." In both models wakeup-sets are chosen in the order in which their tags arrive in the scheduler.

Set-distributed wakeup is analogous to arranging the wakeup-sets into linked lists, where only the head of a linked list can be examined in a cycle. Set-first is the array structure analog, which represents the ability to examine the entire wakeup-set in a cycle. Set-first wakeup is the more ideal of the two policies since it can allow for more instructions to be awakened per

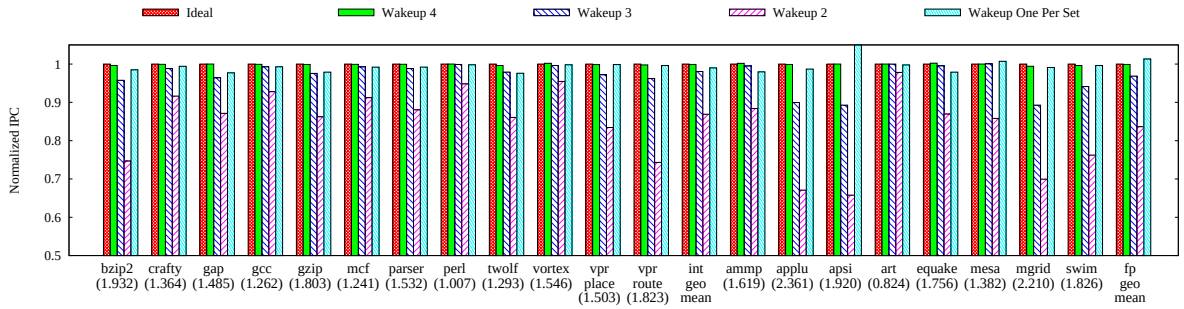


Figure 3.1 Limited Wakeup Bandwidth, Base Configuration

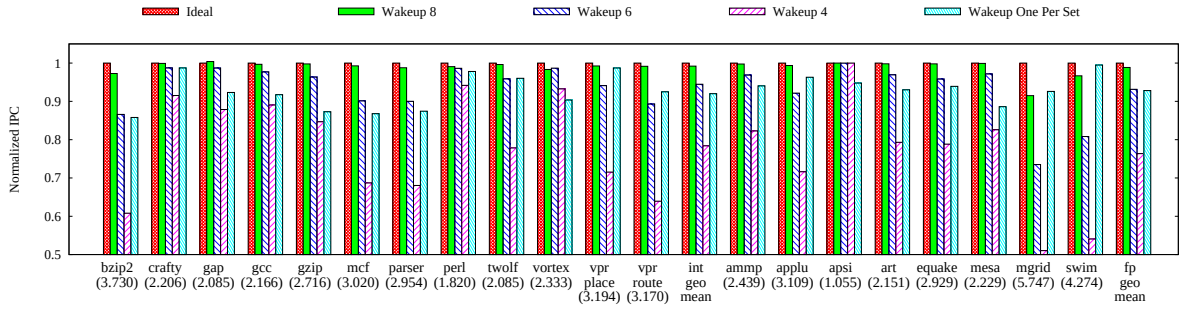


Figure 3.2 Limited Wakeup Bandwidth, Aggressive Configuration

cycle than set-distributed wakeup. If there are fewer wakeup sets than the wakeup bandwidth, set-distributed wakeup is only allowed to look one instruction per set in a cycle. This has the effect of limiting the wakeup bandwidth to the number of tags which have yet to wake up all of their instructions.

Note that we model a penalty for examining an empty wakeup-set, assuming that looking into a set costs one instruction's worth of the wakeup bandwidth. In addition, examining an instruction in a set which has remaining dependencies (it is a member of more than one set) also costs wakeup bandwidth. All instructions are removed from their respective wakeup-sets in fetch order, removing the oldest instructions in a set first.

The results for set-first wakeup compared to the ideal configuration are given in Figures 3.1 and 3.2. Data points labeled "Wakeup B" represent set-first wakeup where B is the bandwidth of wakeup. Results are given for limiting the wakeup bandwidth to the processor width and below for both the base and aggressive configurations. For each benchmark the IPC is normalized to the Ideal configuration. The IPC of the Ideal is given in parentheses below the benchmark name. The geometric mean of the SPECINT and SPECFP benchmarks is also given.

Limiting bandwidth to the width of the machine reduces performance over the ideal by less than 0.15% for the SPECINT and SPECFP benchmarks in the base configuration. For the

aggressive machine the SPECINT benchmarks show a 0.8% degradation in performance over the ideal, while the SPECFP benchmarks degrade by 1.16%. As wakeup bandwidth is reduced below the native pipeline bandwidth, the performance degrades further.

An additional data point, "Wakeup One Per Set," is given in Figures 3.1 and 3.2, which represents set-distributed wakeup *with no bandwidth constraint.* In other words, this data point represents the ability to examine and awaken one instruction per set for all wakeup-sets each cycle (if they were arranged in linked-lists, for example). This point is given to highlight the significant performance loss in set-distributed wakeup, and the importance of being able to wake up more than one instruction per set in a cycle. The performance loss due to restricting wakeup to one instruction per wakeup-set in the aggressive configuration is near 8% for SPECINT and SPECFP on average. The base configuration performance for set-distributed wakeup degrades by 1.01% for SPECINT and 0.90% for SPECFP (ignoring `apsi`) compared to the ideal.

Figures 3.1 and 3.2 show that there are benchmarks which perform better under wakeup bandwidth limitations than with the ideal scheduler. The increase in performance is generally negligible and can be attributed to effects of having a slightly different wakeup order than the ideal. Benchmarks which achieve higher IPC over the ideal while bandwidth limited generally do so by means of slightly improved branch resolution times, stall behavior, or memory access ordering. An extreme example is `apsi`, where the restrictions placed on wakeup under set-distributed wakeup for the base case cause the 8.33% memory ordering violations which exist in the ideal case to disappear, contributing to the 56% increase in performance.

3.2 Wakeup-Set Structured Scheduling

Making use of the fact that bandwidth can be limited without adversely affecting system performance, we propose building a scheduler which divides the structures involved with instruction wakeup from those performing instruction selection and issue. The general design, shown in Figure 1.1, p. 2, allows for instructions which enter the scheduler which have not had all of their dependencies satisfied to be stored in wakeup-sets associated with the tags which will awaken them. Dispatched instructions from renaming which are ready to issue are placed directly into the ready-instruction queue bypassing the wakeup-sets array. Instructions move to the ready-instruction queue from the wakeup-sets as their dependencies are satisfied. Since the rate of wakeup will be less than the rate of arriving tags, a ready-tag buffer is required to hold the tags which have been produced but have not yet awakened all of their dependent

instructions. A more detailed depiction of the wakeup-set scheduler is given in Figure 4.1 and is discussed below.

3.2.1 Ready-instruction queue

The separation of ready and waiting instructions into separate structures is motivated by two factors. The first is that issuing instructions from a set of ready instructions does not require priority encoder or selection logic to cover every instruction in the scheduler, just the set of instructions which are ready to issue. More importantly, it is possible to remove the priority encoder entirely if instructions are issued in the order in which they become ready for execution.

Section 2.3 showed that the number of ready instructions is a small percentage of the total number of instructions contained within the idealized scheduler. This motivates building a small ready-instruction queue separate from the structure which holds nonready instructions. If instructions are issued in FIFO order from the ready-instruction queue, the priority encoder can be removed from the issue logic and selection only need examine as many instruction entries as there are issue ports, greatly simplifying selection.

3.2.2 Wakeup-sets array

The second factor which motivates the structural separation of instructions with dependencies from those which are ready to issue is the desire to eliminate the associative search through all instructions in the scheduler at each tag production. Wakeup can scale as the size of the scheduler increases without affecting performance by limiting the set of instructions each physical tag is required to examine.

The separation of the ready from the waiting instructions allows instructions to be stored in wakeup-sets for this purpose, as motivated by Section 2.5. Wakeup-sets are, in general, small and do not grow significantly with scheduler size. If wakeup-sets are implemented as limited sized arrays, limiting their size to four should achieve the majority of the performance of unlimited sized wakeup-sets based on the results shown in Figure 2.4. The wakeup-sets array is depicted in Figure 4.2. There are four instructions per set, each tagged with a valid (V) bit and a bit to determine additional dependences, discussed in Section 4.2.

Limiting the size of the wakeup-sets requires buffering those instructions which cannot fit into their designated wakeup-set. An instruction must be able to be inserted into all of

its wakeup-sets in order for it to be placed in the wakeup-sets array structure. When an instruction dispatches which cannot allocate space in a desired wakeup-set it must diverted to a *set-overflow queue*, shown in Figure 4.1. Instructions are inserted and removed in FIFO order from the set-overflow queue. An instruction can be removed from the head of the set-overflow queue only when its dependencies have become satisfied or there is space in all of its remaining wakeup-sets. Instructions are removed from the set-overflow queue by making use of an extra scoreboard port, to determine if the instruction dependencies have been satisfied, and an extra insertion port into the wakeup-set structure.

Since the ready-instruction queue is of limited size, both the process of wakeup and removing instructions from the set-overflow queue must be stalled if the ready-instruction queue runs out of entries. It is the case that there is enough bandwidth to insert instructions into the ready-instruction queue from rename, wakeup, and the set-overflow queue without need to prioritize between them as long as there is space in the ready-instruction queue.

3.2.3 Ready-tag buffer

Since the bandwidth of wakeup is limited, more resource tags may be waiting to wake up their dependents than there is bandwidth to wake them up. Therefore the physical tags are tracked in the *ready-tag buffer* structure as they are transmitted to the scheduler. For the purposes of our design we assume the ready-tag buffer to be implemented as a queue of tags. A bitmap, one entry per tag, could also be used to mark which tags are ready to awaken their dependencies.

CHAPTER 4
EVALUATION

Section 4.1 shows the performance of limiting the wakeup bandwidth and the number of instructions in a wakeup-set, concluding that neither limitation adversely effects performance significantly. However, upon close examination, the ability to insert instructions into more than one wakeup-set comes at the cost of increased circuit complexity, which does not significantly outweigh the performance gain. Therefore in Section 4.2 we explore the performance of a "single wakeup-set scheduler" which inserts individual instructions into at most one wakeup-set at dispatch.

We then show in Section 4.3 that a single wakeup-set scheduler implemented with a banked wakeup-sets array gives performance which is within 5% of our highly idealized instruction scheduler with 1024 entries, and within 2% of the performance of a ideal 64 entry instruction scheduler, in our aggressive configuration. This configuration is what we propose as the most implementable, realistic configuration of the wakeup-set scheduler. Our final configuration is shown in Figure 4.1.

4.1 Multiple Wakeup-Set Insertion

Our physical implementation of the wakeup-set structured scheduler is constrained in the number of instructions which belong to a wakeup-set. Thus there will naturally be a performance difference when compared to the results in Chapter 3 due to the delay in waking up instructions which do not fit into the limited sized wakeup sets and are thus buffered in the set-overflow queue.

The bandwidth of wakeup is limited to four and eight instructions in the base and aggressive configurations, respectively. The size of the ready-instruction queue is 32 entries for the base and 128 entries for the aggressive configuration. It is assumed that an instruction can awaken and be selected for issue within the same cycle. Wakeup-sets are limited to four instructions in both configurations. These sizes are chosen based on the data gathered from the idealized instruction scheduler studies in Chapters 2 and 3.

By separating the scheduler into a wakeup-sets array and ready-instruction queue we create a scheduler which can effectively hold `P x W + R` total instructions, where `P` is the number

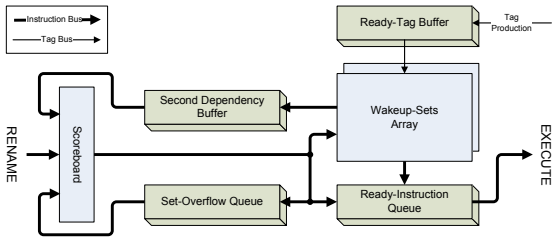


Figure 4.1 Single Wakeup-Set Scheduler

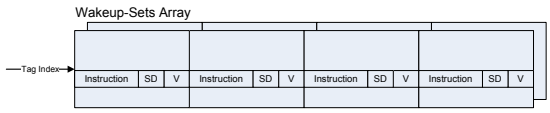


Figure 4.2 Wakeup-Sets Array

Each entry in the wakeup-sets array contains an instruction, a valid (V) bit, and a second dependency (SD) bit.

of physical tags, W is the size of a wakeup-set, and R is the size of the ready-register queue. This is due to organizing the wakeup-sets as a tag-indexed array of wakeup-sets as shown in Figure 4.2.

Results are given in Figures 4.3 and 4.4 for limiting the size of each wakeup set to four. This configuration is labeled "Multi Set," indicating that instructions are placed into as many wakeup-sets as they have dependencies. These simulations, and those in the following sections, are done with a set-overflow buffer limited to 8 entries in the base configuration and 32 in the aggressive. Sizing the set-overflow buffer as such results in performance nearly identical to that of having infinitely sized buffers.

The performance loss due to limiting the wakeup-set size to four in the aggressive configuration is 0.90% for SPECINT, which is marginally worse than the 0.80% performance loss seen from limiting the wakeup bandwidth alone. The aggressive configuration has a performance loss of 0.96% over the ideal for SPECFP. In the base configuration the performance depredation is less than 0.08% overall due to limiting wakeup-set size to four.

We consider this an acceptable degradation in performance considering that we are comparing to a very large, ideal instruction scheduler. The "Ideal" scheduler in the base case has 128 entries, whereas the aggressive ideal scheduler has 1024 entries which is unimplementable by prior means. Note that in Figures 4.3 and 4.4 we provide an additional "Ideal" data point which represents the base and aggressive ideal instruction scheduler limited to 32 and 64 entries, respectively. When compared to the smaller ideal schedulers the multiple wakeup-set scheduler

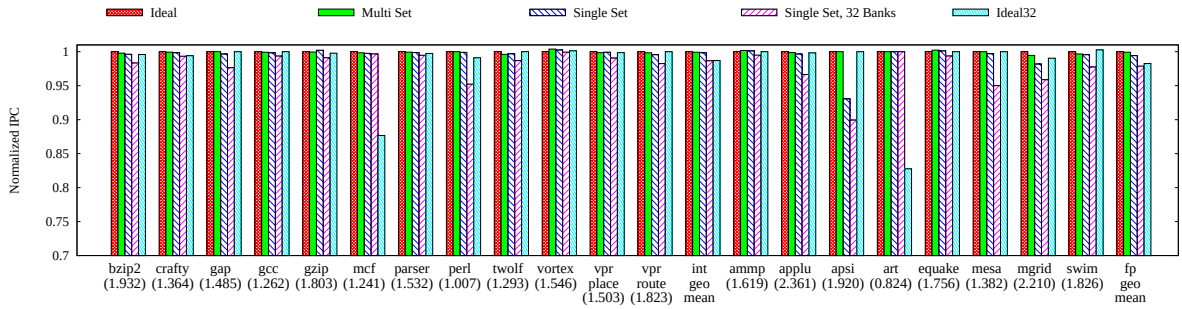


Figure 4.3 Wakeup-Set Size Limited, Base Configuration

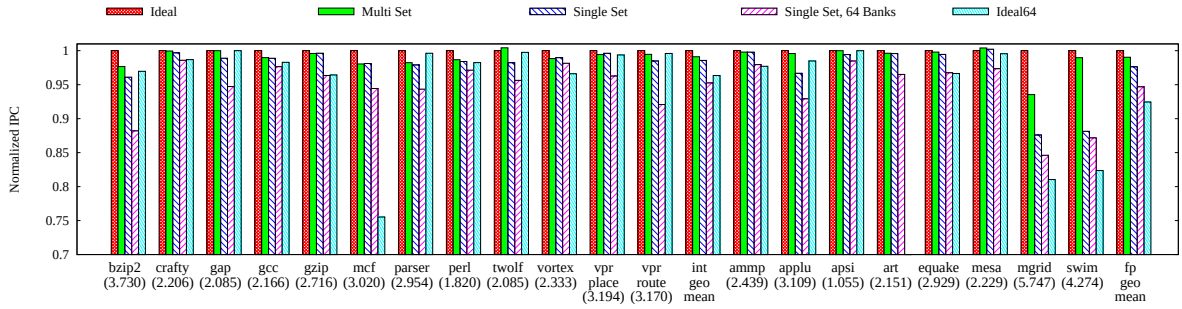


Figure 4.4 Wakeup-Set Size Limited, Aggressive Configuration

performs, on average, 2.87% better for the SPECINT and 7.12% better for the SPECFP benchmarks in the aggressive configuration. We consider this an encouraging result considering the complexity of building a scheduler of these sizes by standard techniques.

4.2 Single Wakeup-Set Insertion

Inserting an instruction into more than one wakeup-set at dispatch increases the hardware complexity of the wakeup-set scheduler without providing a significant performance advantage. The complication primarily comes from increasing the number of ports required to insert an instruction with multiple dependencies at dispatch. An instruction requires a port per wakeup-set that it attempts to insert itself in, potentially doubling the number of ports required for dispatching instructions. Additionally, wakeup is further complicated by the need to determine if a single instruction is present in more than one wakeup-set for each instruction awakened.

Handling instructions with more than one dependency as a special case is a viable option since the number of instructions dispatched with additional dependencies is small. Under the base configuration 12.2% (16.2% for the aggressive) of all instructions dispatched have two

outstanding dependencies, taking the geometric mean across all benchmarks simulated. In total, the base and aggressive configurations have 61.1% and 75.1% percent instructions, respectively, which are not immediately ready to issue at dispatch and are thus sent to the wakeup-sets array.

For an instruction with more than one dependency we randomly pick a wakeup-set from its two outstanding dependencies to place the instruction in. The instruction is marked with a *second dependency (SD) bit* to indicate that the instruction has an additional outstanding dependency. The second dependency bit is used at the time of wakeup to trigger a scoreboard check on the second dependency.

If, upon wake up, the second dependency has been satisfied the instruction is steered to the ready-instruction queue. If the instruction has a remaining dependency, it is placed in the *second dependency buffer*, shown in Figure 4.1. The second dependency buffer is used to hold instructions which need to be reinserted into the wakeup-sets array to await another tag production. The scoreboard is duplicated to supply the ports required to check these dependencies at wakeup (not shown in Figure 4.1).

It is possible that an instruction's dependencies are satisfied before it can be removed from the second dependency buffer and placed into the wakeup-sets array. Therefore instruction dependencies are checked in the scoreboard before reinsertion into the wakeup-sets array. If there are no remaining dependencies for the instruction, it is directed toward the ready-instruction queue.

The second dependency buffer cannot be a blocking FIFO queue if deadlock is to be avoided. It is possible that an instruction cannot be moved from the second dependency buffer to the wakeup-sets array or the ready-instruction queue. If all of an instruction's dependencies have been satisfied but the ready-instruction queue is full, the instruction will remain in the second dependency buffer until such time as there is room in the ready-instruction queue.

If there exists a second dependency but the wakeup-set for the tag is full, the instruction cannot be placed into its wakeup-sets array. Blocking on the second dependency buffer until the tag is produced will result in deadlock since the instruction may block the producer of the tag it is waiting on. Similarly, moving the instruction into the in-order, head blocking set-overflow buffer can potentially deadlock the machine.

Therefore the second dependency buffer is implemented as a small associative buffer. Instructions which cannot be moved to the ready-instruction queue or their wakeup-set remain in the second dependency buffer until such time as they can be removed. Only one instruction

can be removed from the second dependency buffer per cycle. In the base configuration the size of the second dependency buffer is 4 entries, 32 entries in the aggressive configuration.

Figures 4.3 and 4.4 show the results for single wakeup-set insertion, labeled "Single Set." The base configuration does not see an appreciable difference in performance due to single wakeup-set insertion. The aggressive configuration IPC is affected slightly by inserting instructions into only one wakeup-set; the geometric mean of the SPECINT and SPECFP degrading by 0.05% and 1.43%, respectively. This is considered an acceptable performance depreciation in order to reduce the number of ports required on the wakeup-sets array by at least half, and simplify the logic involved in waking up instructions.

4.3 Ports and Banking

The wakeup-sets array will require several ports for the purposes of dispatch and instruction wakeup each cycle. In order to satisfy the ability to issue and awaken the superscalar width of the machine there must be at least two ports into the wakeup-sets per instruction of bandwidth for the single wakeup-set scheduler. To support the wakeup bandwidth of the machine, a port per instruction awakened is required in the worst case scenario where there is only one instruction per wakeup set available.

One port is required for instruction dispatch per instruction which comes from rename. Note that a port per wakeup-set insertion is required at dispatch, making the single wakeup-set configuration less complicated than inserting instructions into two wakeup-sets per instruction. We simulated assuming two additional instruction dispatch ports into the wakeup-sets array for the purposes of inserting instructions into the wakeup-sets array from the set-overflow buffer and second dependency buffer.

Both the scoreboard and ready-instruction queue are assumed not to incur an additional cost from adding ports. The scoreboard can be easily replicated to support the required ports. The ready-instruction queue is an in-order FIFO that can be split into individual queues, one per write port, if necessary.

In order to support the number of ports in an efficient manner we bank the wakeup-sets array, showing the performance effects of modeling the port requirements. Figure 4.3 shows the effect of implementing 32 banks for the base single wakeup-set configuration. Figure 4.4 shows the effects of implementing 64 banks for the single wakeup-set aggressive configuration.

Banking the base configuration results in a modest degradation of performance, reducing the IPC by nearly 2% for both SPECINT and SPECFP. Compared to the single wakeup-set

insertion model, banking in the aggressive configuration reduced performance by less than 3.36% for the SPECINT and SPECFP benchmarks. The reduction in performance from banking the aggressive configuration compared to the ideal is 4.74% over SPECINT and 5.31% for SPECFP.

We consider this configuration of the wakeup-set scheduler as the most valid point of comparison with respect to the ability of implementation in hardware. With this in mind we consider these performance losses when compared to an ideal a very positive result. Moreover, compared to the 64 entry ideal instruction scheduler, the banked aggressive configuration suffers a performance loss of only 1.13% for the SPECINT and a performance *improvement* of 2.42%. Given the sizes of instruction schedulers in current processors, and the infeasibility of constructing a scheduler of this size, this comparison highlights the utility of the wakeup-set structured scheduler as a replacement for current "issue buffer" implementations of the out-of-order instruction scheduler structure.

4.4 Checkpointed Processor Misprediction

Checkpointed processor implementations, such as the Alpha 21264, introduce the need to cancel only a portion of the instructions inside the scheduler in the event of a misprediction. In noncheckpointed processors the entire scheduler is flushed of instructions via gang invalidation. In a checkpointed processor, misspectulation results in the need to remove only those instructions fetched after the misspeculated instruction.

In the wakeup-set structured scheduler this means that each active wakeup-set modified after the dispatch of the misspectulated instruction must be examined and those instructions which are within the range of canceled instructions must be invalidated. Additionally, canceled instructions must also be removed from the set-overflow queue, the second dependency buffer, and the ready-instruction queue.

4.5 Evaluation Summary

In Section 4.3 we presented a banked wakeup-set scheduler which inserts instructions into only one wakeup-set at dispatch and has reasonably implementable port requirements. The design of this scheduler lends itself to scalability such that we accept a modest performance loss when compared to a highly idealized instruction scheduler of an unimplementable size by conventional means. When compared to a scheduler of a smaller, yet still considerably large size, the wakeup-set structured scheduler performs within a marginal performance difference.

Thus we believe the wakeup-set structured scheduler to be a viable option for future high-performance out of order microprocessors.

CHAPTER 5
BACKGROUND AND RELATED WORK

The wakeup-sets approach taken here is most closely related to previous approaches based on direct matching. Weiss and Smith's Direct Tag Search algorithm [4] is the earliest direct matching based dynamic scheduling scheme that we are aware of. The Direct Tag Search algorithm keeps wakeup-sets of width at most one and handles overflow by stalling dispatch. The Monsoon Explicit Token-Store architecture also used direct matching [5]. The Explicit Token-Store kept wakeup-sets of width 2, and overflow was handled by the compiler. In particular, the compiler recorded the consumers of each producer in the producer instruction. If an instruction had more than two consumers then the compiler inserted explicit "fan-out" nodes.

Önder and Gupta's Direct Wake-up approach stores information similar to a wakeup-set as a linked list [6, 7]. They deal with wakeup-sets of size larger than two by building use-use chains of dependent instructions. Canal and González take a different approach in their N-use scheme [8, 9]. The N-use scheme includes an array of wakeup-sets, each of maximum width N, and instructions that overflow their wakeup-sets are scheduled in-order. Both the Direct Wake-up approach and the N-use scheme handle instructions waiting for multiple producers by placing those instructions in multiple wakeup sets.

Our wakeup-sets design extends these earlier direct matching approaches in two important ways. First, we have demonstrated that, rather than inserting an instruction into multiple wakeup-sets, it is sufficient to insert the instruction into only one of its wakeup-sets, and then check the additional dependency at wakeup. This reduces the porting requirements of the wakeup-sets array by a factor of two. Second, we have demonstrated that the wakeup-sets array can be banked rather than multiported.

It is also instructive to compare the wakeup-sets approach to avoiding associative matching to the approaches used in directory based cache-coherent shared-memory multiprocessors to avoid broadcasting coherence messages. By analogy to an associative scheduler where each tag is broadcast to all waiting instructions, a snoopy bus-based coherence protocol will broadcast each invalidation address on a bus to all other caches in the system [10]. Directory based cache-coherence protocols, on the other hand, can scale to much larger bandwidths because they store the set of caches that have a copy of each cache-line in a directory [11]. Rather than broadcast invalidation messages, point-to-point messages can be sent to just those caches that

actually need to be involved in the transaction. By analogy to limiting the width of the wake-up set, "limited" directory based cache-coherence schemes [12] restrict the maximum size of the set they can store to some fixed value.

A different approach to scaling scheduling windows takes advantage of the fact that instruction completion latencies are highly predictable [8, 9, 13–15]. If one knows the times at which producer tags will be ready, it is easy to predict the time at which a consumer instruction will wake up (i.e., on the cycle that the last producer will produce its tag). The problem with this approach, as pointed out by Brown et al., is that while completion latencies are highly predictable, contention for selection slots is less predictable [16, 17]. Because the contention rate is relatively low, however, one can assume that all instructions are selected on the same cycle that they are woken up and later check the prediction. This completely removes the latency of selection from the crictical scheduling loop.

Another approach to reducing scheduling window latency involves "slicing" the dataflow graph. The earliest example of a slicing scheduler is the complexity-effective design of Palacharla et al. [1]. They point out that, given a chain of dependent instructions, only the head of the chain need be considered for wakeup. More recently this idea has been extended to large window machines specifically to avoid blocking the processor pipeline on long-latency cache misses [18–20].

The most common microarchitectural approach to scaling the scheduler in out-of-order implementions is to cluster instructions based on instruction type at dispatch. The AMD Athlon™XP processor has a split scheduler with 18 entries in its integer scheduler and twice as many entries in the floating point scheduler [21]. Dividing the instruction scheduler based on instruction type is not unique to the Athlon™. The Intel Pentium™4 divides its instructions into two FIFO scheduling queues at dispatch, one for memory operations and another for integer/floating point operations [22]. The IBM POWER5™architecture has four issue queue types on which it clusters [23]. The fixed point and memory operation queues have 36 entries, while the floating point queue has 24, branch execution instructions have 12, and logical condition register instruction queue has 10 entries.

CHAPTER 6
CONCLUSIONS AND FUTURE WORK

Scaling the size of the out-of-order instruction scheduler is difficult in conventional implementations primarily due to the instruction wakeup and selection logic critical path. Separating at dispatch instructions which are ready to issue from those which are waiting to be awakened enables the design of a scheduler which takes advantage of this separation.

The most important property which we discover is that peak wakeup bandwidth does not need to be higher than the dispatch/issue bandwidths to support instruction throughput in the instruction scheduler. We also observe that the majority of instructions in the scheduler are awaiting wakeup, and are not ready to issue. Additionally, if a scheduler can precompute wakeup-sets as instructions are dispatched, the number of instructions analyzed to perform wakeup on one tag is considerably small. We show that the wakeup-set scheduler takes advantage of these properties in order to allow for higher performance and scalability. The wakeup-set scheduler can achieve the performance of an aggressive highly idealized out-of-order instruction scheduler.

This work assumes that the wakeup-sets array can be constructed with as many sets as the number of dependency tags. This leads to exceedingly large wakeup-sets arrays, particularly in the presence of store-set prediction which will add several dependency tags that the scheduler must be able to track. Given that the number of wakeup-sets which are active on average is considerably smaller than the number of physical tags in our system, one can conceive of ways to reduce the number of wakeup-sets via hashing into a smaller wakeup-sets structures. Our simulations do not model the delay in moving an instruction from wakeup to the issue structure, a potentially important design consideration considering the size of the wakeup-sets array. We leave these as subjects for future study.

Additionally, inserting instructions into a single wakeup-set at dispatch introduces the choice as to which set to place the instruction in. We chose this set randomly, but it could prove to be easily predicted to improve performance. Introducing a store-set predictor and the associated dependencies that must be enforced could possibly make this decision an important area of exploration for future work.

REFERENCES

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *ISCA '97: Proceedings of the 24th International Symposium on Computer Architecture*, 1997, pp. 206–218.

[2] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *ISCA '01: Proceedings of the 28th International Symposium on Computer Architecture*, 2001, pp. 74–85.

[3] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint processing and recovery: Towards scalable large instruction window processors," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003, p. 423.

[4] S. Weiss and J. E. Smith, "Instruction issue logic in pipelined supercomputers," *IEEE Transactions on Computers*, vol. 33, pp. 1013–1022, Nov. 1984.

[5] G. M. Papadopoulos and D. E. Culler, "Monsoon: An explicit token-store architecture," in *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 82–91.

[6] S. Önder and R. Gupta, "Superscalar execution with dynamic data forwarding," in *PACT '98: Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, 1998, pp. 130–135.

[7] S. Önder and R. Gupta, "Instruction wake-up in wide issue superscalars," in *Euro-Par '01: Proceedings of the 7th European Conference on Parallel Computing*, 2001, pp. 418–427.

[8] R. Canal and A. González, "A low-complexity issue logic," in *Proceedings of the 2000 International Conference on Supercomputing*, May 2000, pp. 327–335.

[9] R. Canal and A. González, "Reducing the complexity of the issue logic," in *Proceedings of the 2001 International Conference on Supercomputing*, June 2001, pp. 312–320.

[10] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *ISCA '83: Proceedings of the 10th International Symposium on Computer Architecture*, June 1983, pp. 124–131.

[11] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Transactions on Computers*, vol. C-27, pp. 1112–1118, Dec. 1978.

[12] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An evaluation of directory schemes for cache coherence," in *ISCA '88: Proceedings of the 15th International Symposium on Computer Architecture*, May 1988, pp. 280–289.

[13] P. Michaud and A. Secznec, "Data-flow prescheduling for large instruction windows in out-of-order processors," in *HPCA 7: Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan. 2001, pp. 27–36.

[14] S. E. Raasch, N. L. Binkert, and S. K. Reinhardt, "A scalable instruction queue design using dependence chains," in *ISCA '02: Proceedings of the 29th International Symposium on Computer Architecture*, 2002, pp. 318–329.

[15] D. Ernst, A. Hamel, and T. Austin, "Cyclone: A broadcast-free dynamic instruction scheduler with selective replay," in *ISCA '03: Proceedings of the 30th International Symposium on Computer Architecture*, 2003, pp. 253–263.

[16] J. Stark, M. D. Brown, and Y. N. Patt, "On pipelining dynamic instruction scheduling logic," in *MICRO 33: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 2000, pp. 57–66.

[17] M. D. Brown, J. Stark, and Y. N. Patt, "Select-free instruction scheduling logic," in *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 2001, pp. 204–213.

[18] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A large, fast instruction window for tolerating cache misses," in *ISCA '02: Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[19] R. D. Barnes, E. M. Nystrom, J. W. Sias, S. J. Patel, N. Navarro, and W. Hwu, "Beating in-order stalls with "flea-flicker" two-pass pipelining," in *MICRO 36: Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, 2003, p. 387.

[20] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 107–119.

[21] J. Huynh, "The AMD Athlon™XP processor with 512KB L2 cache," Feb. 2003.

[22] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium™4 processor," *Intel Technology Journal*, vol. 5, no. 1, 2001.

[23] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "POWER5 system microarchitecture," *IBM Journal of Research and Development*, vol. 49, no. 4/5, 2005.