Select-Free Instruction Scheduling Logic

Mary D. Brown † Jared Stark ‡ Yale N. Patt †

Dept. of Electrical and Computer Engineering †
The University of Texas at Austin
{mbrown,patt}@ece.utexas.edu

Microprocessor Research Labs ‡
Intel Corporation
jared.w.stark@intel.com

Abstract

Pipelining allows processors to exploit parallelism. Unfortunately, critical loops—pieces of logic that must evaluate in a single cycle to meet IPC (Instructions Per Cycle) goals—prevent deeper pipelining. In today's processors, one of these loops is the instruction scheduling (wakeup and select) logic [10]. This paper describes a technique that pipelines this loop by breaking it into two smaller loops: a critical, single-cycle loop for wakeup; and a non-critical, potentially multi-cycle, loop for select. For the 12 SPECint 2000 benchmarks, a machine with two-cycle select logic (i. e., three-cycle scheduling logic) using this technique has an average IPC 15% greater than a machine with three-cycle pipelined conventional scheduling logic, and an IPC within 3% of a machine of the same pipeline depth and one-cycle (ideal) scheduling logic. Since select accounts for more than half the scheduling latency [10], this technique could significantly increase clock frequency while having minimal impact on IPC.*

1. Introduction

Processor performance has increased a thousandfold over the past twenty years. Much of this increase is due to deeper pipelines, which enable greater exploitation of parallelism. Over these twenty years, pipeline depths have grown from 1 (Intel® 286 processor), to 5 (Intel486™ processor), to 10 (Intel Pentium® Pro processor), to 40[1] (Intel Pentium 4 processor) [6]. Pipeline depths will continue to grow as processors attempt to exploit more parallelism.

Obstacles to pipelining are critical loops [2]. These are pieces of logic that must evaluate in a single cycle to meet IPC (Instructions Per Cycle) performance goals. An example of a critical loop—and the subject of this work—is the wakeup and select (i. e., dynamic instruction scheduling) logic [10]. The wakeup logic determines when instructions are ready to execute, and the select logic picks ready instructions for execution. Because instructions cannot wake up until all instructions they are dependent on have been selected, the wakeup and select logic form a critical loop. If this loop is stretched over more than one cycle, dependent instructions cannot execute in consecutive cycles. For the SPECint2000 benchmarks, IPC is 10% lower when stretching this loop over two cycles, and 19% lower when stretching it over three cycles.

This paper introduces a technique that pipelines the scheduling loop by breaking it into two smaller loops: one, critical, single-cycle loop for wakeup; and one, non-critical, potentially multi-cycle loop for select. This is accomplished by speculating that all waking instructions are immediately selected for execution. We call this technique *select-free scheduling*. The select logic—no longer in a critical loop—can use intelligent, IPC-boosting priority schemes without impacting cycle time.

The paper describes several examples of implementations of this technique for a generic dynamically scheduled machine. Experimental results for the 12 SPECint2000 benchmarks show that, for two-cycle select logic (i. e., three-cycle scheduling logic), a machine using this technique has an average IPC 15% greater than a machine with three-cycle pipelined conventional scheduling logic, and an IPC within 3% of a machine of the same pipeline depth and one-cycle (ideal) scheduling logic. For one-cycle select logic, a machine using this technique has an IPC 8.9% greater than a machine with two-cycle pipelined conventional scheduling logic, and an IPC 0.7% less than a machine of the same pipeline depth and one-cycle (ideal) scheduling logic. Since select accounts for more than half the scheduling latency [10], this technique could signifi-

[1] The pipeline depth is given in terms of fast clocks. The fast clock cycle time was set equal to the time required to execute an operation and bypass its result to its dependent operations. [5]

cantly increase clock frequency while having only a minimal impact on IPC.

In addition to increasing the clock frequency, select-free scheduling can also be used to enlarge the scheduling window and/or reduce the window's power consumption. If a machine's conventional scheduling logic is replaced with select-free scheduling logic, but the clock frequency is held constant rather than increased, the window can be enlarged. Or, the window can be built from slower, lower-power transistors. The machine's design constraints (power, frequency, IPC) dictate how select-free scheduling is used.

2. Related Work

Weiss and Smith [12] provide an introduction to schedulers. They describe four scheduling paradigms: Tomasulo's algorithm, Thornton scoreboarding, direct tag store, and in-order execution. For more current information, Yeager [13] describes the scheduler for the MIPS* R10000* microprocessor. Farrell and Fischer [3]; and Chandrakasan, Bowhill, and Fox [2]; describe the scheduler for the Compaq* Alpha* 21264 processor. And Hinton et al. [5] describe the scheduler for the Intel Pentium 4 processor.

Recently, Palacharla, Jouppi, and Smith [10] presented a detailed analysis of the delays in the scheduler and compared it to the delays in other parts of a modern processor. They concluded that the scheduler, along with the operand bypass, are likely to be the most critical paths as the machine width and frequency are increased.

To prevent the scheduler from becoming the critical path, researchers have investigated three classes of techniques: preschedulers, low-latency schedulers, and pipelined schedulers. Many of these techniques, as well as the technique described in this paper, may be used in combination with one another.

Michaud and Seznec [7] proposed prescheduling instructions before they are written into the scheduling window. The prescheduler takes sequentially ordered instructions and arranges them in dataflow order so that they pass quickly through the scheduling window. With prescheduling, a large effective scheduling window can be built using a relatively small, low-latency, scheduler.

Several researchers proposed low-latency schedulers. Palacharla, Jouppi, and Smith [10] proposed placing chains of dependent instructions into FIFOs, and issuing from multiple FIFOs to functional units in parallel. They expect this scheduler to have lower latency than a conventional scheduler due to its relative simplicity. Önder and Gupta [9] described a scheduler that limits the number of dependent instructions that a selected instruction can immediately wakeup. By limiting this number, they hope the scheduler is smaller and faster than a conventional scheduler. Henry et al. [4] described how to build faster schedulers using cyclic segmented prefix circuits. Canal and González [1] described two schedulers that eliminate most of the associative look-up logic that is used by conventional schedulers. They believe that eliminating associative look-up logic is the key to building low-latency schedulers.

Finally, our previous paper [11] described *speculative wakeup*, which pipelines the scheduling logic over two cycles while having only a minor impact on IPC. Speculative wakeup uses a dependency lookahead scheme to stretch the critical scheduling loop (wakeup + select) over two cycles while still allowing dependent instructions to schedule in consecutive cycles. Select-free scheduling is complementary to speculative wakeup; it removes the select logic from this critical loop. These two techniques can be combined for even more extreme pipelining of the scheduling logic.

3. Machine Model

3.1. Pipeline Overview

The baseline processor is a conventional superscalar out-of-order processor. Figure 1 shows its pipeline. Instructions are fetched and decoded in the first two stages. The rename stage translates architectural register identifiers into physical register identifiers. The scheduler is responsible for issuing instructions to the execution units when all required resources (source operands and execution units) are available. The remainder of the pipeline consists of the register file read, execute/bypass, and retirement stages. Some stages of the pipeline may require more than one cycle.

Figure 1. Processor Pipeline

3.2. The Execution Core

Figure 2 shows the main structures that make up the execution core. These structures and the core operation will be discussed briefly.

The Rename Stage. The rename stage assigns a new physical register to every destination operand, and maps the source operands of subsequent instructions onto the corresponding physical registers. The renamer determines whether the source operands needed by an instruction currently reside in the physical register file, or whether the instruction needs to wait for another instruction in the scheduling window to produce the operand. The rename logic outputs the physical register number of the source operand. As the rename logic determines dependences in

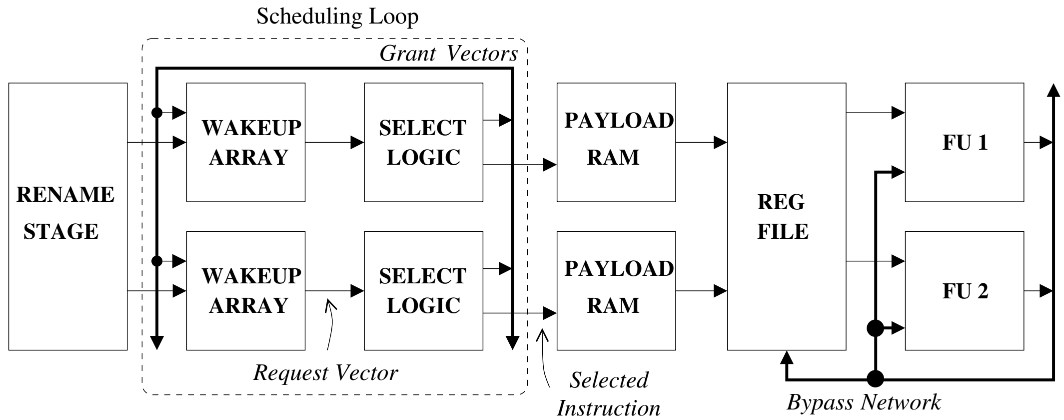

Figure 2. Execution Core

terms of physical register numbers, it also calculates these dependences in terms of scheduler entry numbers.

The Wakeup Arrays. After instructions are renamed, they are placed in the scheduling window. The scheduling window consists of one or more wakeup arrays, each feeding a separate selector (i. e., piece of select logic). The wakeup logic monitors the resource dependencies for instructions to determine if they are ready to execute. The wakeup logic sends the select logic a vector (the Request Vector) indicating which instructions are ready for execution.

The Select Logic. The select logic picks one instruction from those marked in a Request Vector for execution on a given functional unit. In conventional microprocessor designs, the select logic contains a prioritizer which typically picks the oldest instructions from the Request Vector. The select logic outputs a vector (the Grant Vector) indicating the selected instructions which in turn becomes the input to the wakeup logic in the next clock. This input wakes up the instructions' dependents. Hence the scheduling logic is a loop: instructions that are ready to be scheduled in the current clock produce results which are fed to dependent instructions that must be scheduled in the following clock (or some number of clocks in the future depending on execution latency). The need to prioritize all ready instructions adds to the delay in the scheduling loop; by removing the prioritization and performing selection in a subsequent stage, this loop can be made to run at a significantly higher frequency.

It is possible to design select logic that can pick multiple instructions per cycle to execute on multiple functional units. Both the Compaq Alpha 21264 processor [2] and the MIPS R10000 microprocessor [13] use schedulers that can pick 2 instructions per cycle. A distributed scheduling window, where each functional unit has a separate scheduler and wakeup array, will have the fastest schedulers because the wakeup arrays are small and the select logic must only pick 1 instruction per cycle. Unified (or semi-unified) scheduling windows, which use one scheduler to pick several instructions per cycle, may be slower, but they eliminate the load balancing problems present in distributed scheduling windows.

Payload RAM and Register File. After an instruction is selected for execution, the instruction's *payload* is obtained from a table. The payload is information needed for the instruction's register file access and execution such as its opcode and the physical register identifiers of its sources and destination. [3]

Execution and Scheduling Window Deallocation. Some time after an instruction has been granted execution, it is deallocated from the wakeup array. It remains in the instruction window until it retires, however. By holding only a subset of the instructions from the instruction window in the wakeup arrays, the wakeup arrays can be built smaller, which will reduce the scheduling latency. When the instruction is deallocated, the rename mapper is updated to indicate that the instruction's dependents should get its result from the register file rather than the bypass network.

Scheduling Implications of Using Heterogeneous Functional Units. An instruction must be steered to a functional unit that can execute it. Consequently, it must be steered to a scheduler feeding that type of functional unit.

It may be advantageous to build machines with two classes of functional units: low-latency functional units and high-latency functional units. Fast schedulers are needed for the low-latency functional units, and slow schedulers can be used for the high-latency functional units, as was done on the Intel Pentium 4 processor [5]. The fast and slow schedulers may have the same total scheduling latency (i. e., the time required for both wakeup and select). What differentiates the two types of schedulers is the time between when a scheduler schedules an instruction and when the instruction's dependents can be scheduled; that is, the latency of the scheduler's critical loop. This latency is lower for the fast scheduler than for the slow scheduler. In general, the latency of a scheduler's critical loop must be less than or equal to the latency of the functional unit it feeds in order to prevent the insertion of bubbles into the execution pipeline.

An exception to this rule occurs with instructions that do not produce register results, such as most branches. Since these instructions have no instructions that depend on them via registers, the latency of the critical loop in the scheduler has absolutely no impact on performance. Hence, these instructions can safely be scheduled using a slow scheduler.

Another exception occurs with instructions that do not produce critical results. These instructions might also be scheduled using a slow scheduler. Slow schedulers may also be used to save power or allow larger scheduling windows.

In our machine model, the fast and slow schedulers have the same number of pipeline stages, and the same total scheduling latency. The slow schedulers are used for branches and high-latency instructions, and the fast schedulers are used for the remaining low-latency instructions.

4. Baseline Scheduling Logic

The scheduling logic is comprised of wakeup arrays, selectors, and countdown timers.

Each wakeup array entry contains the wakeup logic for a single instruction. Our implementation uses wire-OR-style wakeup logic [3, 8] instead of traditional CAM-style wakeup logic, although either style could be used with select-free scheduling logic. Each entry contains a bit vector, called a *Resource Vector*, that indicates which resources the instruction needs. Each bit position, or *Resource Bit*, within this vector corresponds to a particular resource. A resource can be either a result operand produced by the instruction in a particular entry of a wakeup array, or a particular functional unit[2]. Each Resource Bit is set if the instruction requires that resource, and reset if it doesn't.

Figures 3 and 4 show a dependency graph and an example of a wakeup array that contains the instructions in the graph. The portion of the wakeup array that is shown has four Resource Vectors with seven Resource Bits. The instructions in entries 1–4 are the SHIFT, SUB, ADD, and MULT instructions from the dependency graph. In this example, the instructions that produced the values for the unspecified source operands of the SHIFT, SUB, ADD, and MULT instructions have already executed, so their result values reside in the register file. The SHIFT instruction only requires the shifter, so only one Resource Bit is set. The SUB and ADD instructions depend on the result of the SHIFT and require the ALU, and the MULT instruction depends on the result of the SUB and requires the multiplier.

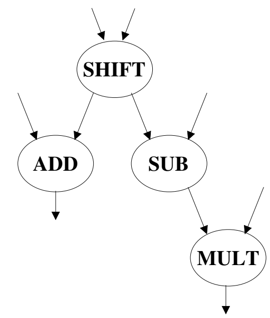

Figure 3. Dependency Graph

		Functional Unit Required			*Result Required From ...*				
		SHIFTER	**ALU**	**MULTIPLIER**	**ENTRY 1**	**ENTRY 2**	**ENTRY 3**	**ENTRY 4**	
(SHIFT)	**ENTRY 1**	1							• • •
(SUB)	**ENTRY 2**		1		1				• • •
(ADD)	**ENTRY 3**		1		1				• • •
(MULT)	**ENTRY 4**			1		1			• • •

Figure 4. Wakeup Array

Figure 5 shows the wakeup logic for one wakeup array entry. The AVAILABLE lines running vertically pass through every entry in the array. Each line corresponds to a Resource Bit in the Resource Vector. The line is high if the resource is available and low if it is not. The SCHEDULED bit indicates whether or not the instruction has been granted execution. There may be a number of cycles between the time the instruction is granted execution and the time it is actually deallocated from the wakeup array. During this time, the SCHEDULED bit is set to prevent the instruction from requesting execution again. If the instruction must be rescheduled, for example, due to a load latency misprediction [8], its SCHEDULED bit is reset by asserting the Reschedule line. The instruction requests execution if (1) its SCHEDULED bit is not set, and (2) for each resource, it does not require that resource or that resource is available. The AND gate is implemented using a wire-OR structure to make it fast. Hence, we call this style of wakeup logic wire-OR-style.

Each selector is a priority circuit. Its input is a bit vector indicating which instructions from the wakeup array re-

[2] If a particular functional unit can begin executing a new instruction every cycle, then a Resource Bit is not needed for that functional unit.

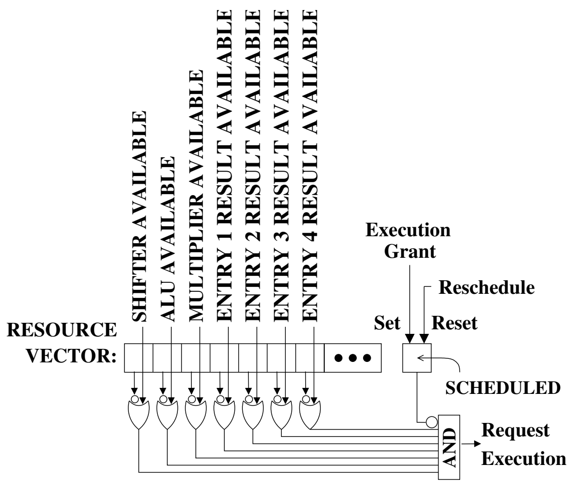

Figure 5. Logic for One Wakeup Array Entry

quest execution. One of its outputs is the Grant Vector, indicating which instructions receive the execution grants. The wakeup array uses this Grant Vector to set the SCHEDULED bits. The other outputs are a set of one-hot bit vectors. The first one-hot specifies the first instruction that received an execution grant. The second one-hot specifies the second instruction that received an execution grant. And so on. For a select-1 priority circuit, there is only 1 one-hot, and it is equivalent to the Grant Vector. Each one-hot is used to access a port (or the port) of a Payload RAM and deliver the payload for the associated instruction to the register file and to the functional unit. The one-hot is the Payload RAM's set of word lines, so the Payload RAM doesn't have (or require) word decoders.

After instructions receive execution grants, the AVAILABLE lines for their wakeup array entries are asserted so that their dependent instructions may wake up. For a single-cycle instruction, the AVAILABLE line is asserted immediately. For an N-cycle instruction, the AVAILABLE line is asserted $N-1$ cycles later. This is accomplished by using a countdown timer initialized to the instruction's latency. When an instruction receives an execution grant, its timer begins to count down. When the timer reaches 1, it asserts the instruction's AVAILABLE line.

With wire-OR wakeup logic, data dependencies are specified in terms of wakeup array entries rather than physical register identifiers. When an instruction's wakeup array entry is deallocated, it may still have dependent instructions residing in the wakeup arrays. In order to prevent an incorrect dependence on a new instruction that gets allocated to the same entry, when the entry is deallocated, every wakeup array entry in the scheduling window clears the Resource Bit that corresponds to the deallocated entry.

5. Select-Free Scheduling Logic

This section explains the operation and implementation of select-free scheduling logic. For simplicity, we will only discuss an implementation for single-cycle instructions, although it is also possible to implement select-free schedulers for multi-cycle instructions. Section 5.1 provides the rationale for select-free scheduling. Section 5.2 explains some terminology. Section 5.3 explains the implementation of the scheduling pipeline. Section 5.4 discusses two ways of avoiding incorrect schedules.

5.1. Rationale

In a given wakeup array, usually no more than one instruction becomes ready per cycle. Simulations show that a 16-entry wakeup array in a machine with 8 select-1 schedulers has, on average, no waking instructions in 53% of the cycles (including branch recovery cycles), one waking instruction in 39% of the cycles, and two or more waking instructions in the remaining 8% of the cycles.[3] Because there is usually no more than one instruction per wakeup array requesting execution, it is possible to speculate that any waking instruction will be selected for execution. Select-free scheduling logic exploits this fact by removing the select logic from the critical scheduling loop and scheduling instructions speculatively. The select logic is only used to confirm that the schedule is correct.

5.2. Collisions and Pileups

With select-free scheduling logic, instructions speculate that they will be selected for execution, and they assert the AVAILABLE lines for their wakeup array entries before their selection is really confirmed.

A *collision* is the scenario where more instructions wake up than can be selected, resulting in a misspeculation. Any unselected instructions assert their AVAILABLE lines too early. We will call these instructions the *collision victims*. Collision victims are identified at the same time an instruction is selected: when the Grant Vector is produced, a second vector of collision victims is also produced. Dependents of the collision victims may wake up before they are really ready to be scheduled, thus entering the scheduling pipeline too early. We call these instructions *pileup victims*. Pileup victims are identified by a scoreboard check before the execute stage. The misspeculation is analogous to a freeway accident with the instructions being the vehicles. Hence the terms *collision* and *pileup*. The next section explains in more detail how select-free scheduling logic detects and reschedules collision and pileup victims.

[3] Section 6 provides the machine configurations.

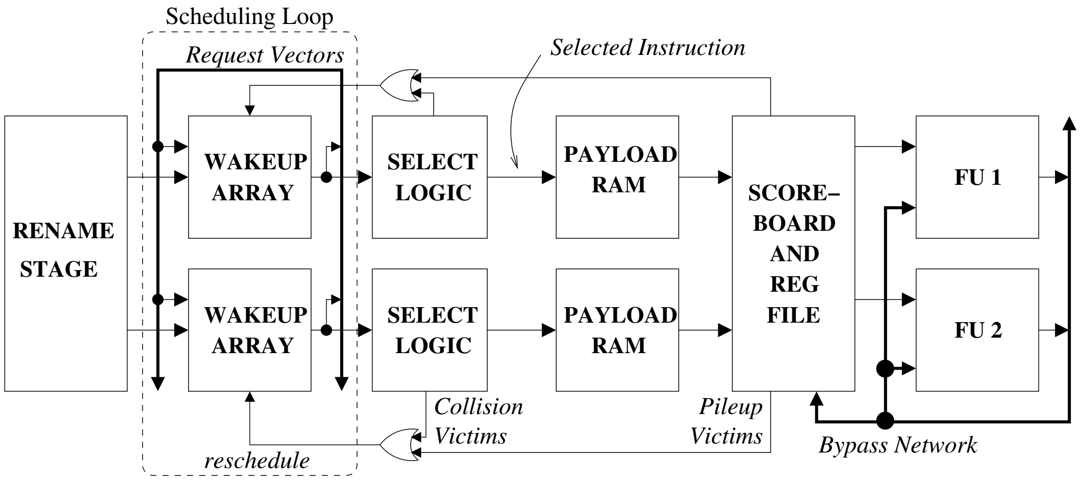

Figure 6. Execution Core with Select-Free Scheduling

5.3. Select-Free Scheduling Implementation

Figure 6 shows an execution core using a select-free scheduler. The wakeup logic is the same as for baseline scheduling logic shown in Figure 5. With select-free scheduling, an instruction assumes it will be selected when it wakes up. Hence, once an instruction is awakened, it immediately sets its SCHEDULED bit and asserts its AVAILABLE line. The select logic still produces a set of one-hot vectors that are used to index the Payload RAM, but it also produces a Collision Victim vector indicating which requesting instructions did *not* receive an execution grant. An instruction that is selected is not necessarily really ready for execution because it may have been a pileup victim.

To check for pileup victims, a scoreboard is placed after the Payload RAM. The scoreboard is accessed in parallel with the register file, and does not add pipeline stages. The scoreboard records which instructions have been correctly scheduled. An instruction reads the scoreboard to determine if the instructions that produce its sources have been correctly scheduled. If all have been correctly scheduled, the instruction records in the scoreboard that it was correctly scheduled. Otherwise, the instruction is a pileup victim and does not update the scoreboard.

When an instruction is identified as a collision or pileup victim, the SCHEDULED bit of its wakeup array entry must be reset so that it will be rescheduled. A bit vector specifying the pileup victims is ORed with the Collision Victim vector produced by the select logic to indicate which Reschedule lines must be asserted. Instructions should not be deallocated from the wakeup entries until they have passed the scoreboard check.

With select-free scheduling logic, collision and pileup victims suffer a scheduling penalty. With the baseline scheduling logic, an instruction may request execution every cycle until it is granted execution. With select-free scheduling logic, a collision victim—or any misscheduled instruction—takes at least as long as the latency of the scheduling pipeline to reschedule. Pileup victims will incur an additional penalty if the payload RAM and scoreboard logic are pipelined over several cycles.

When a machine uses a combination of baseline and select-free schedulers, only the select-free schedulers will have collision victims. However, the AVAILABLE lines originating from the select-free schedulers pass through the baseline schedulers, and may be speculative. Therefore, pileup victims may reside in either type of scheduler. Hence instructions from all schedulers must check the scoreboard.

5.4. Collision Avoidance Techniques

This section describes two ways to avoid collisions.

Select-N Schedulers. One way to avoid collisions is to use schedulers that can select more than one instruction per cycle. For select-1 schedulers, there is a collision when 2 or more instructions request execution. For select-2 schedulers, there is a collision when 3 or more instructions request execution. As the number of instructions selected increases and the total number of schedulers decreases, the probably of a collision decreases. To demonstrate this, three machines were simulated, each with eight functional units and the same size scheduling window. The first had eight select-1 schedulers, the second had four select-2 schedulers, and the third had two select-4 schedulers. For an average cycle, the probability of a collision in any scheduler for the machine with select-1 schedulers was 39%, for the machine with select-2 schedulers was 26%, and for the machine with select-4 schedulers was 15%. Although select-2 and select-4 logic are more complex than select-1 logic, select-free scheduling allows this logic to be pipelined with little loss in IPC.

Predict Another Wakeup (PAW). If an instruction can determine that another instruction in the same wakeup array will wake up at the same time as itself, it can avoid a collision by delaying its execution request. For example, the ADD in Figure 7 will wakeup at the same time as the SUB. If the ADD's wakeup logic knows that this will happen, it can delay its request. However, dynamically determining that this will happen is quite complex. It is easier to detect that an instruction in the same wakeup array might *potentially* wake up. This is accomplished by detecting when an operand required by any older instruction becomes available. This detection is made using a bit vector, called the *PAW vector*, that indicates all of the sources needed by all older instructions in the wakeup array.

Like the Resource Vector, each bit position of the PAW vector corresponds to a particular resource. A bit is set if there is an older instruction in the wakeup array requiring the resource. Figure 8 shows an example of the wakeup array with PAW vectors for the instructions in the dependency graph in Figure 7. The 1s indicate the bits of the Resource vectors that are set. The shaded portions indicate the bits of the PAW vectors that are set.

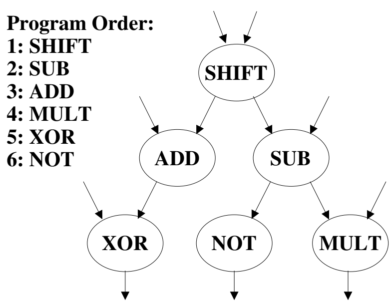

Figure 7. Dependency Graph

Instructions should *not* request execution when any of the resources marked in the PAW vector first become available. Each time one of these resources becomes available, an awake instruction delays its execution request one cycle. For example, the ADD will not request execution the first cycle that the ENTRY 1 (SHIFT's) AVAILABLE line is asserted even though it will be ready to execute, because the SUB also wakes up this cycle. If the ADD's AVAILABLE line becomes asserted the cycle after the SUB's AVAILABLE line becomes asserted, the NOT will delay its request for 2 cycles after it wakes up.

Although PAW vectors reduce the collision rate by over 50%, they increase the amount of state stored in the wakeup array and possibly the size and/or latency of the wakeup array. The wakeup logic for each entry is also modified: the request line can only be asserted if the bitwise AND of the PAW Vector and the AVAILABLE lines is zero. The PAW Vector bits must be reset when the AVAILABLE lines are first asserted to prevent further delays in requesting execution.

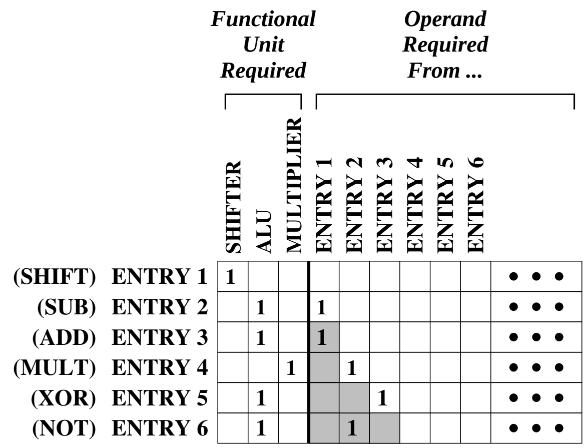

Figure 8. Wakeup Array

The PAW vector for each instruction is computed in the rename stage as follows: The rename stage has one register per wakeup array. This register is the PAW vector for the next instruction to be placed in that array. The instruction first reads the register to determine its PAW vector, and then updates the register by ORing the register with the portion of the instruction's Resource Vector marking the instruction's own source operands. When AVAILABLE lines are first asserted, the resource bits corresponding to those lines are cleared from the register.

The PAW technique does not perfectly predict when older instructions request execution. One reason is that many instructions have two sources. When the first source of an instruction with two sources becomes available, the instruction does not wake up, although it may prevent a younger, waking instruction from requesting execution due to a PAW vector match. Another reason is that two instructions may both delay their execution requests because of a PAW vector match, and then both request execution in the following cycle, resulting in a collision. Despite these problems, the PAW technique is still good enough to make a noticeable improvement in IPC.

6. Experiments

All experiments were run using a cycle-accurate simulator for the Alpha ISA. The SPECint2000 benchmarks were compiled using the Compaq C Compiler with aggressive optimizations (the `-fast` switch and feedback-directed optimizations). All benchmarks were run to completion using modified input sets to reduce run time.

Branch Predictor	16-bit gshare, 4096-entry BTB
Instruction Cache	64KB 4-way set associative (pipelined) 2-cycle directory and data access
Instruction Window	256 instructions
Scheduling Window	128 instructions
Execution Width	4 single-cycle functional units + 4 pipelined, multi-cycle functional units
Data Cache	64KB 4-way set associative (pipelined) 2-cycle directory and data access
Unified L2 Cache	1MB, 8-way, 7-cycle access 2 banks, contention is modeled
Main Memory	100 cycles minimum access

Table 1. Machine Configuration

Instruction Class	Latency (in Cycles)
integer arithmetic	1
integer multiply	8, pipelined
fp arithmetic	4, pipelined
fp divide	16
loads and stores	1 + dcache latency
all others	1

Table 2. Instruction Latencies

All experimental machines were 8-wide superscalar processors, configured as shown in Table 1. Instruction latencies are shown in Table 2. To prevent instruction re-execution from becoming a factor in our analysis, all machines used perfect load latency prediction and memory dependence prediction. Each machine required 2 cycles for fetch, 2 for decode, 2 for rename, 1 for Payload RAM read, 1 for register read, and 1 for retire. All machines used 1-cycle wakeup logic and either 1-cycle or 2-cycle select logic. Each machine had four pipelined functional units for executing multi-cycle instructions (including loads and stores) and branches, and four functional units for executing single-cycle instructions. On average, 52% of the instructions were executed on the multi-cycle functional units. An instruction with a 1-cycle execution latency required a minimum of 11 or 12 cycles (depending on the select logic latency) to advance from the first fetch stage to retire.

6.1. Scheduler Configurations

To measure the effectiveness of select-free scheduling, we modeled several machines: a baseline machine, six machines with select-free scheduling, and an ideal machine. The six machines with select-free scheduling are divided into three pair, with each pair using a different technique for handling collisions. Within each pair, one machine uses the PAW technique and the other doesn't.

The **baseline** machine uses baseline scheduling logic for all functional units, except that it pipelines this logic over two or three cycles. As a result, there is a minimum of two cycles between the scheduling of an instruction and its dependent instructions. This means there is at least a one cycle bubble between the execution of a single-cycle instruction and its dependents, although independent instructions may be scheduled to fill these bubbles. There are no bubbles between a multi-cycle instruction and its dependents.

The three pair of machines with select-free scheduling have select-free schedulers for the fast functional units, and pipelined baseline schedulers for the multi-cycle functional units. The first pair, labeled **Scoreboard** and **Scoreboard, PAW** in the graphs, use the implementation described in Section 5; that is, collision and pileup victims reset their SCHEDULED bits when they are in the Collision Victim vector or they fail the scoreboard check. The second pair of machines (**Squash All**) squash and reschedule all instructions from all stages of the scheduling pipeline whenever the last stage of select detects a collision. The third pair of machines (**Squash Dep**) only squash collision victims and their dependents from the scheduling pipeline as soon as a collision is detected. The simulator performs dependency analysis to determine which instructions should be squashed. The second and third pair of machines are not suggested implementations, but they show the worst and best case rescheduling techniques for the described scheduling logic implementation. Neither of these pair require a scoreboard because they have no pileup victims.

The **ideal** machine pipelines the scheduling logic over the same number of cycles as the other machines, but still allows dependents of single-cycle instructions to schedule without pipeline bubbles. Conceptually, this machine is a machine with 1-cycle scheduling logic and one or two extra pipeline stages. The pipeline depth was kept consistent with the other machines to remove the effects of changing the branch mispredict penalty. Because the wakeup/select loop is performed in one cycle, its clock frequency is considerably lower than that of the **baseline** machine or the machines with select-free scheduling.

6.2. Simulation Results

Figure 9 shows the harmonic means of the IPC on the SPECint2000 benchmarks of all eight machines using both 1-cycle and 2-cycle select logic. All machines have eight select-1 schedulers. The bars showing the IPC of the machines using 1-cycle and 2-cycle select logic are overlaid; the upper bars (in black) show the IPC for the machines with 1-cycle select logic.

Of the select-free machines, the **Squash All** machines have the lowest IPC because correctly scheduled instructions may be squashed unnecessarily. The simultaneous re-waking of these squashed instructions caused further collisions. The **Squash Dep** machines had the highest IPC of the select-free machines because pileup victims were squashed

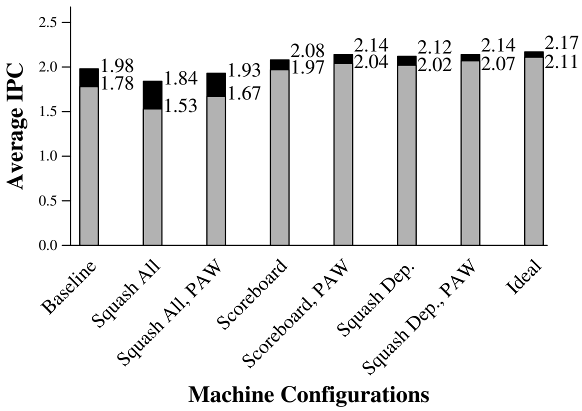

Figure 9. Average IPC

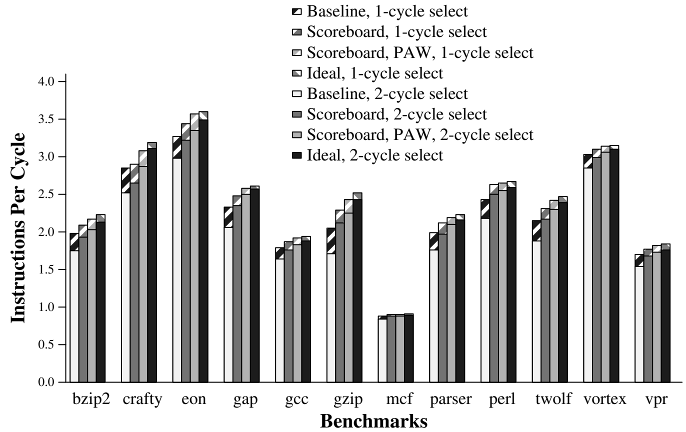

Figure 10. IPC with 8 Select-1 Schedulers

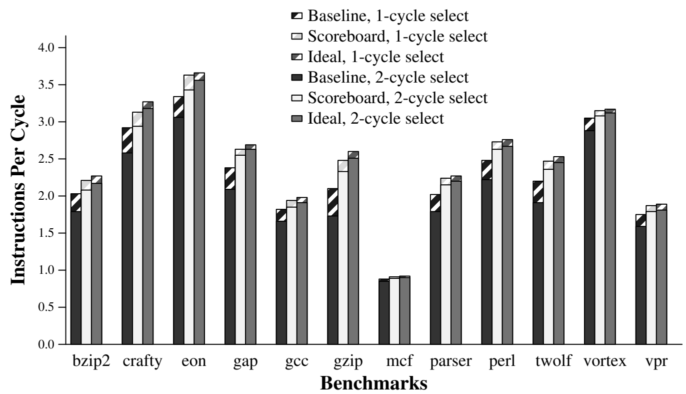

Figure 11. IPC with 4 Select-2 Schedulers

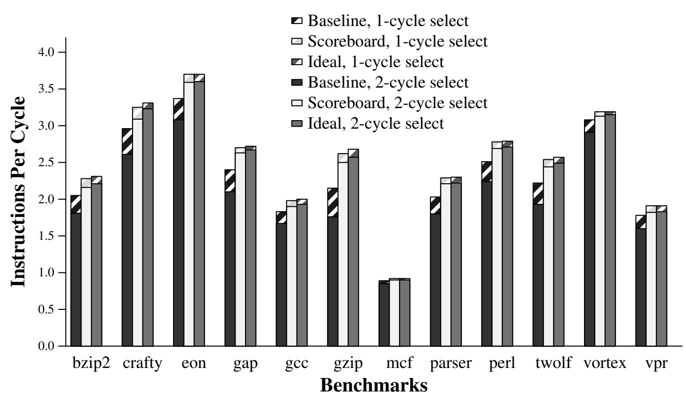

Figure 12. IPC with 2 Select-4 Schedulers

when collisions were detected, allowing the independent instructions to be selected for execution.

The **Baseline** machine had the largest IPC difference when moving from 1-cycle to 2-cycle select logic because of the extra pipeline bubble between a single-cycle instruction and its dependents. The machines with select-free scheduling logic and 2-cycle select performed worse than their counterparts with 1-cycle select primarily because the pileups were larger, causing more instructions to be rescheduled after a collision. The IPC difference between the two **Ideal** machines reflects only the increased branch misprediction penalty.

Simulations of all machines were run using 8, 4, and 2 schedulers that each selected 1, 2, and 4 instructions per cycle, respectively. The number of wakeup array entries per scheduler were 16, 32, and 64, respectively. Hence the total size of the scheduling window was always 128 instructions. Figures 10, 11 and 12 show the IPC for the benchmarks using select-1, select-2, and select-4 schedulers. Variations on the PAW technique were only marginally effective for machines with select-2 and select-4 schedulers. Hence we show results for PAW only for select-1 schedulers.

The average IPC and fraction of retired instructions that are collision victims and pileup victims are shown in Table 3. The machines which squash instructions when collisions are detected have no pileup victims.

The PAW technique reduces the number of collision victims by over half for the select-1 schedulers. While this technique is more effective at reducing collisions than using select-2 schedulers, the IPC improvements with each technique are similar. This is because the PAW technique may unnecessarily delay ready instructions from waking up.

7. Conclusion

The wakeup and select logic is a critical loop in high-performance processors. Select-free scheduling breaks this loop into two smaller loops: a critical single-cycle loop for wakeup, and a non-critical (potentially) multi-cycle loop for select. By breaking this loop, the processor cycle time is no longer set by the time required for wakeup and select, but is instead set just by the time required for wakeup.

The benefit provided by this aggressive pipelining of

Machine Model	IPC	collision victim	pileup victim
1-cycle select			
Baseline, Select-1	1.98	n/a	n/a
Baseline, Select-2	2.01	n/a	n/a
Baseline, Select-4	2.03	n/a	n/a
Squash All, Select-1	1.84	12.9%	n/a
Squash All, Select-1, PAW	1.93	9.5%	n/a
Squash All, Select-2	1.96	9.1%	n/a
Squash All, Select-4	2.07	6.4%	n/a
Scoreboard, Select-1	2.08	10.6%	11.0%
Scoreboard, Select-1, PAW	2.14	4.9%	3.8%
Scoreboard, Select-2	2.18	7.3%	7.2%
Scoreboard, Select-4	2.23	4.4%	3.7%
Squash Dep, Select-1	2.12	10.3%	n/a
Squash Dep, Select-1, PAW	2.14	4.9%	n/a
Squash Dep, Select-2	2.19	6.6%	n/a
Squash Dep, Select-4	2.23	4.0%	n/a
Ideal, Select-1	2.17	n/a	n/a
Ideal, Select-2	2.22	n/a	n/a
Ideal, Select-4	2.24	n/a	n/a
2-cycle select			
Baseline, Select-1	1.78	n/a	n/a
Baseline, Select-2	1.81	n/a	n/a
Baseline, Select-4	1.83	n/a	n/a
Squash All, Select-1	1.53	13.9%	n/a
Squash All, Select-1, PAW	1.67	9.6%	n/a
Squash All, Select-2	1.67	10.4%	n/a
Squash All, Select-4	1.81	7.2%	n/a
Scoreboard, Select-1	1.97	10.0%	13.4%
Scoreboard, Select-1, PAW	2.04	4.9%	6.6%
Scoreboard, Select-2	2.09	7.0%	9.1%
Scoreboard, Select-4	2.15	4.3%	5.3%
Squash Dep, Select-1	2.02	9.2%	n/a
Squash Dep, Select-1, PAW	2.07	4.4%	n/a
Squash Dep, Select-2	2.12	6.4%	n/a
Squash Dep, Select-4	2.16	9.2%	n/a
Ideal, Select-1	2.11	n/a	n/a
Ideal, Select-2	2.15	n/a	n/a
Ideal, Select-4	2.17	n/a	n/a

Table 3. Fraction of Retired Instructions that are Collision Victims or Pileup Victims

the wakeup and select logic depends on the type of processor you want to design. If you want narrow-issue and high-frequency, the aggressive pipelining allows you to build deep pipelines. If you want wide-issue and low-frequency, the aggressive pipelining allows you to build a large scheduling window. If you want low-power, the aggressive pipelining allows you to build your scheduling window out of slower, lower-power transistors. And, if you want wide-issue and high-frequency and low-power (Good Luck!), the aggressive pipelining allows you to build a deeply pipelined processor with a large scheduling window built from low-power transistors.

Acknowledgements

We specially thank Andy Glew, Shih-Lien Lu, Ed Grochowski, Robert Chappell, Paul Racunas, and the anonymous referees for their insights and comments on this work. In addition, we gratefully acknowledge the other members of the HPS and MRL research groups for the stimulating environments they provide. This work was supported in part by Intel and IBM. Mary Brown is supported by an IBM Cooperative Graduate Fellowship.

References

[1] R. Canal and A. González. A low-complexity issue logic. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 327–335, 2000.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox, editors. *Design of High-Performance Microprocessor Circuits*. IEEE Press, 2001.

[3] J. A. Farrell and T. C. Fischer. Issue logic for a 600-MHz Out-of-Order execution microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5), 1998.

[4] D. S. Henry, B. C. Kuszmaul, G. H. Loh, and R. Sami. Circuits for wide-window superscalar processors. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 236–247, 2000.

[5] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technical Journal*, Feb. 2001. Q1 2001 Issue.

[6] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2001.

[7] P. Michaud and A. Seznec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, pages 27–36, 2001.

[8] E. Morancho, J. M. Llabería, and À. Olivé. Recovery mechanism for latency misprediction. In *Proceedings of the 2001 ACM/IEEE International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[9] S. Önder and R. Gupta. Superscalar execution with dynamic data forwarding. In *Proceedings of the 1998 ACM/IEEE Conference on Parallel Architectures and Compilation Techniques*, pages 130–135, 1998.

[10] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[11] J. Stark, M. D. Brown, and Y. N. Patt. On pipelining dynamic instruction scheduling logic. In *Proceedings of the 33th Annual ACM/IEEE International Symposium on Microarchitecture*, 2000.

[12] S. Weiss and J. E. Smith. Instruction issue logic in pipelined supercomputers. *IEEE Transactions on Computers*, C-33(11):1013–1022, Nov. 1984.

[13] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–41, Apr. 1996.