On Pipelining Dynamic Instruction Scheduling Logic

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

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

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

Abstract

A machine's performance is the product of its IPC (Instructions Per Cycle) and clock frequency. Recently, Palacharla, Jouppi, and Smith [3] warned that the dynamic instruction scheduling logic for current machines performs an atomic operation. Either you sacrifice IPC by pipelining this logic, thereby eliminating its ability to execute dependent instructions in consecutive cycles. Or you sacrifice clock frequency by not pipelining it, performing this atomic operation in a single long cycle. Both alternatives are unacceptable for high performance.

This paper offers a third, acceptable, alternative: pipelined scheduling with speculative wakeup. This technique pipelines the scheduling logic without eliminating its ability to execute dependent instructions in consecutive cycles. With this technique, you sacrifice little IPC, and no clock frequency. Our results show that on the SPECint95 benchmarks, a machine using this technique has an average IPC that is 13% greater than the IPC of a baseline machine that pipelines the scheduling logic but sacrifices the ability to execute dependent instructions in consecutive cycles, and within 2% of the IPC of a conventional machine that uses single cycle scheduling logic.

1. Introduction

To achieve higher levels of performance, processors are being built with deeper pipelines. Over the past twenty years, the number of pipeline stages has grown from 1 (Intel 286), to 5 (Intel486), to 10 (Intel Pentium Pro), to 20 (Intel Willamette) [2, 6]. This growth in pipeline depth will continue as processors attempt to exploit more parallelism.

As pipeline depths grow, operations that had previously taken only a single pipeline stage are pipelined. Recently, Palacharla, Jouppi, and Smith [3] stated: "Wakeup and select together constitute what appears to be an *atomic* operation. That is, if they are divided into multiple pipeline stages, dependent instructions cannot issue in consecutive cycles." They use the word *atomic* here to imply that the entire operation must finish before the wakeup/select operations for dependent instructions can begin. Thus, if dependent instructions are to be executed in consecutive cycles—which is necessary for achieving the highest performance—the scheduling logic performs this operation in one cycle.

This paper demonstrates that this logic can be pipelined without sacrificing the ability to execute dependent instructions in consecutive cycles. It introduces *pipelined scheduling with speculative wakeup*, which pipelines this logic over 2 cycles while still allowing back-to-back execution of dependent instructions. With this technique, deeper pipelines and/or bigger instruction windows can be built. This will allow processors to exploit more parallelism, and therefore, allow processors to achieve higher performance.

The paper describes two implementations of pipelined scheduling with speculative wakeup for a generic dynamically scheduled processor: the budget implementation and the deluxe implementation. The budget model has a lower implementation cost than the deluxe model, but not as great an improvement in performance. The generic processor and these two implementations are examples only. There are many processor microarchitectures, and many possible implementations of pipelined scheduling with speculative wakeup. We could not model all of them. Nevertheless, we hope that by examining these simple examples, microarchitects will be able to implement pipelined scheduling with speculative wakeup on real-world microarchitectures.

The paper then compares the IPC (Instructions Per Cycle) of machines using these two implementations to the IPC of a baseline machine that pipelines the scheduling logic but sacrifices the ability to execute dependent instructions in consecutive cycles. For the 8 SPECint95 benchmarks, the average IPC of the machine using the budget implementation is 12% higher than the IPC of the baseline machine, and the IPC of the machine using the deluxe implementation is 13% higher. This paper also compares the IPC of machines using these two implementations to the IPC of a conventional machine that does not pipeline the scheduling logic. Both machines have IPCs that are within 3% of the IPC of the conventional machine. If the critical path through the scheduling logic limits the cycle time for conventional machines, which is very likely if the scheduling operation is considered an atomic unit, these two implementations of pipelined scheduling with speculative wakeup may allow a significant boost in clock frequency with only a very minor impact on IPC.

This paper is divided into six sections. Section 2 presents background information necessary for understanding this study. Section 3 describes conventional instruction scheduling logic. Section 4 describes pipelined scheduling with speculative wakeup. Section 5 presents the experiments, and Section 6 provides some concluding remarks.

2. Background

This section presents background information necessary for understanding our study. Section 2.1 presents our pipeline model. Section 2.2 introduces some terms. Section 2.3 introduces the scheduling apparatus.

2.1. Pipeline Model

Figure 1 shows the pipeline of a generic dynamically scheduled processor. The pipeline has 7 stages: fetch, decode, rename, wakeup/select, register read, execute/bypass, and commit. Each stage may take more than one cycle. For example, the execute/bypass stage usually takes two or more cycles for loads: one cycle to calculate the load address, and one or more cycles to access the cache.

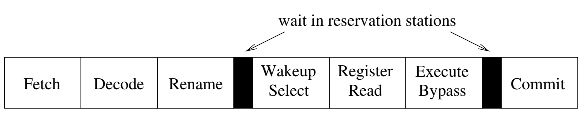


Figure 1. Processor Pipeline

In the fetch stage, instructions are fetched from the instruction cache. They are then decoded and their register operands renamed. Next, they are written into the reservation stations where they wait for their source operands and a functional unit to become available. When this occurs (that is, an instruction wakes up and is selected), the instruction is sent to a functional unit for execution. Its register values are either read from the register file or bypassed from earlier instructions in the pipeline. After it completes execution, it waits in the reservation stations until all earlier instructions have completed execution. After this condition is satisfied, it commits: it updates the architectural state and is deallocated from the reservation stations. [1]

[1] Conventional machines aggressively deallocate reservation stations. We do not consider aggressive deallocation, and simply assume that reservation stations are deallocated at commit.

Note that after an instruction is selected for execution, several cycles pass before it completes execution. During this time, instructions dependent on it may be scheduled (woken up and selected) for execution. These dependent instructions are scheduled optimistically. For example, if they depend on a load, they are scheduled assuming the load hits the cache. If the load misses, the dependent instructions execute—spuriously—without the load result. The dependent instructions must be re-scheduled (and thus, re-executed) once the load result is known.

2.2. Terminology

Figure 2 shows a partial data flow graph. Each node represents an operation. The arrows entering a node represent the values consumed by the operation. The arrow exiting a node represents the value produced by the operation.

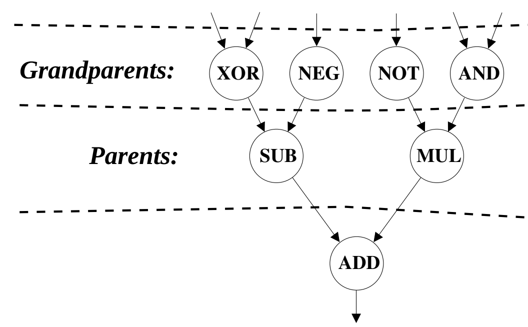


Figure 2. Example Data Flow Graph

The ADD operation consumes the values produced by its *parents*; i. e., the SUB and MUL operations. The ADD's parents consume the values produced by its *grandparents*; i. e., the SUB consumes the values produced by the XOR and NEG operations, and the MUL consumes the values produced by the NOT and AND operations. The reverse relationships also hold: the ADD is the *child* of the SUB and MUL operations; and the *grandchild* of the XOR, NEG, NOT, and AND operations.

2.3. Scheduling Apparatus

Three pieces of logic are needed to perform the dynamic scheduling: rename logic, wakeup logic, and select logic.

The rename logic maps an instruction's architectural register identifiers to physical register identifiers. This eliminates the anti and output register dependencies between instructions. We assume the map is stored in a register file, as described by Palacharla, Jouppi, and Smith [3], and as implemented in the MIPS R10000 [5]. Accessing this register file with an architectural register identifier yields the physical register identifier to which it is mapped.

The wakeup logic is responsible for waking up the instructions that are waiting for their source operands to become available. For conventional scheduling, this is accomplished by monitoring each instruction's parents. For pipelined scheduling with speculative wakeup, it is accomplished by monitoring each instruction's parents *and* grandparents. The wakeup logic is part of the reservation stations. Each reservation station entry (RSE) has wakeup logic that wakes up any instruction stored in it.

The select logic chooses instructions for execution from the pool of ready instructions. We assume each functional unit has a set of dedicated RSEs, as described by Tomasulo [4]. Select logic associated with each functional unit selects the instruction that the functional unit will execute next. The selection is performed by choosing one ready instruction from the functional unit's set of dedicated RSEs.

Figure 3 will be used to further describe the operation of the scheduling apparatus. It shows a microarchitecture that has 8 functional units and 128 RSEs.

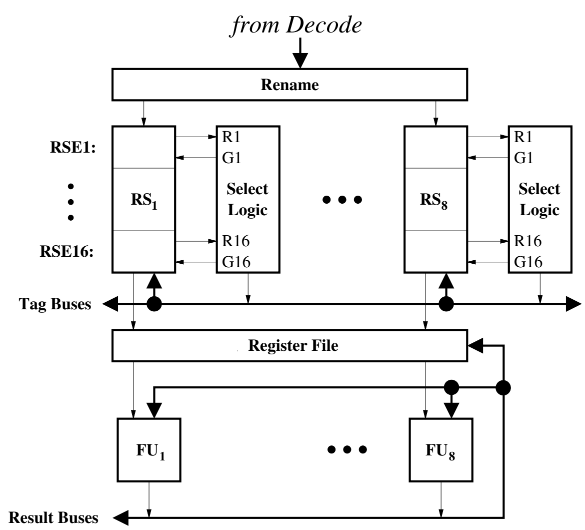


Figure 3. Processor Microarchitecture

Each functional unit has a dedicated set of 16 RSEs, select logic, a tag bus, and a result bus. The select logic chooses the instructions the functional unit executes from the RSEs. After an instruction is chosen, a tag associated with the instruction is broadcast over the tag bus to all 128 RSEs. This tag broadcast signals dependent instructions that the instruction's result will soon be available. After an instruction executes, it broadcasts its result over the result bus to the register file and to any dependent instructions starting execution.

After an instruction is fetched, decoded, and renamed, it is written into a RSE. Each RSE has wakeup logic that monitors the tag buses. For conventional scheduling, when the tags of all the instruction's parents have been broadcast, the RSE asserts its request line. (The request lines are labeled R1–R16.) For pipelined scheduling with speculative wakeup, the RSE asserts its request line, if, for each of the instruction's parents, the parent's tag has been broadcast or all the parent's parents' tags have been broadcast. The select logic for each functional unit monitors the request lines of the functional unit's dedicated set of RSEs, and grants up to one of these requests each cycle. (The grant lines are labeled G1–G16.) After a request is granted, the instruction that generated the request is sent to the functional unit for execution. In addition, the tag for that instruction is broadcast over the tag bus. The instruction either reads its register values from the register file or receives them from instructions just completing execution via bypasses.

3. Conventional Scheduling

Sections 3.1, 3.2, and 3.3 describe the implementations of the rename, wakeup, and select logic for conventional dynamic instruction scheduling. Section 3.4 gives an example of the operation of conventional 1-cycle scheduling, and Section 3.5 gives an example of the operation of conventional scheduling pipelined over 2 cycles.

3.1. Rename Logic

Register renaming performs two primary tasks: allocating physical registers for the destinations of instructions, and obtaining the physical register identifiers for the sources of instructions. An instruction reads the rename map for each architectural source register to obtain the physical register identifier for that source. It also writes the identifier of its allocated physical register into the rename map entry associated with its architectural destination register.

In a superscalar processor, a group of instructions must be renamed at the same time. To detect dependencies between instructions in the same group, the sources of each instruction are compared to the destinations of all previous instructions in the same group. If an instruction's parent is in its group, the identifier of the physical register allocated to the parent overrides the identifier obtained from the rename map. Figure 4 shows the dependency analysis logic for the first three instructions in a group.

3.2. Wakeup Logic

After instructions have been renamed, they wait in reservation stations for their sources to become ready. Each RSE contains information about each of the instruction's sources, such as the physical register identifier (tag) for the source, whether the source is ready, and the number of cycles it takes the producer of the source's value to execute. Figure 5 shows the state information for one RSE. The fields

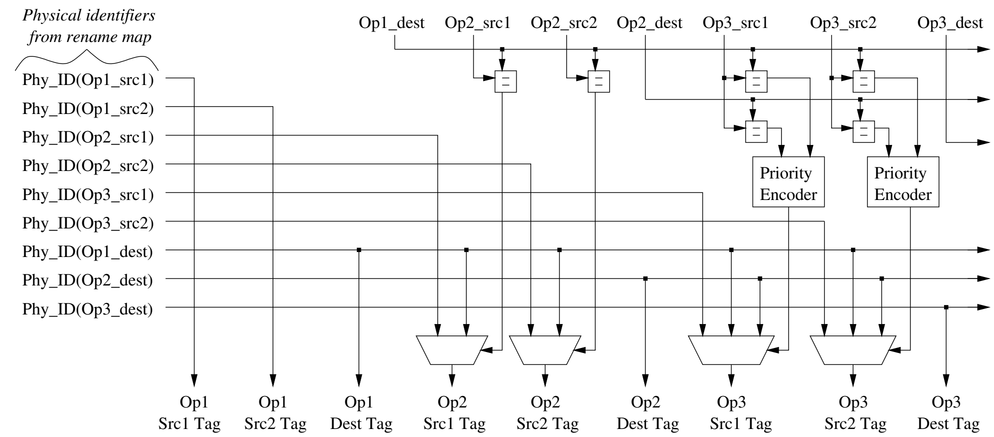


Figure 4. Dependency Analysis Logic for Three Instructions

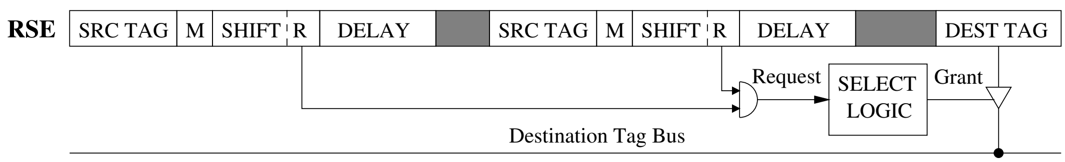


Figure 5. Scheduling Logic for One Reservation Station Entry

labeled SRC TAG contain the tags of the source operands. The R (READY) bit for each source is set if the data for that source is available in the register file or is available for bypass from a functional unit.

In our machine model, instructions broadcast their tags in the same cycle they are selected for execution. Because not all instructions have the same execution latency, the number of cycles between the time their tags are broadcast and the time their results are available is not constant. The DELAY fields are used to handle this variability. [2] For each source, the DELAY field encodes the number of cycles—relative to some base—between when the tag for the source is broadcast and when the associated result is available. We will provide more details shortly about the actual number that is encoded. For the logic implementation described in this paper, this number is encoded as an inverted radix-1 value; e. g., 3 is represented by '1…1000'.

Figure 6 shows the wakeup logic of one source tag for our machine model. It is similar to the MIPS R10000 wakeup logic [5] but has been modified for handling multi-cycle operations. When the destination tag of a parent is broadcast, one of the tag comparators will indicate that a match has occurred, and the M (MATCH) bit will be set. The MATCH bit is a sticky bit that will remain set after the tag match. On a tag match, the SHIFT field is loaded with the value contained in the DELAY field. The SHIFT field is actually contained in an arithmetic right shift register. The MATCH bit is the shift enable for this register. The least significant bit of the SHIFT field is the READY bit mentioned above. After the READY bits for all source operands have been set, the instruction requests execution.

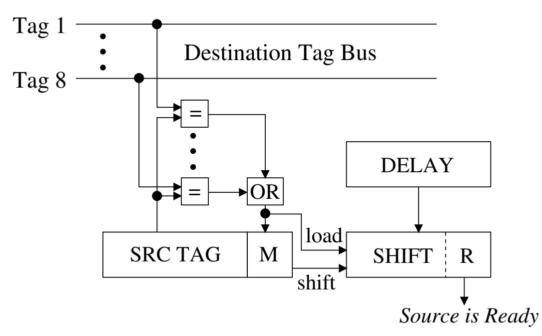


Figure 6. Conventional Wakeup Logic

For a source whose producer has an N-cycle execution latency, the DELAY field contains N-1 zeros in the least significant bits of the field. The remaining bits are all set to 1. This allows the READY bit to be set N-1 cycles after the match. For example, in our model, a load instruction that hits in the data cache takes three cycles to execute. Sup-

[2] Alternative solutions exist. For example, if each functional unit only executes instructions that all have the same latency, the tag broadcast can simply be delayed so that it occurs a fixed number of cycles before the result broadcast. This eliminates the need for the DELAY fields. However, if functional units can execute instructions of differing latencies, this solution is unsuitable at high clock frequencies: Multiple pipe stages may need to broadcast tags, rather than just one. Either the pipe stages will need to arbitrate for tag buses, or the number of tag buses will need to be increased.

pose the DELAY field is four bits. The DELAY field for an instruction dependent on a load would contain the value ‘1100’. When the tag match for the load occurs, this value will be loaded into the SHIFT field, and the MATCH bit will be set. After two more cycles, the SHIFT field will contain the value ‘1111’, and, assuming the load hit in the cache, this source will be ready. For a source operand with a 1-cycle latency, the DELAY field will be ‘1111’. As soon as there is a tag match, the READY bit will be set, allowing the instruction to request execution.

The value for the DELAY field is obtained from a table in the rename stage. The table also provides values for the MATCH bit and SHIFT field for when the tag for a source is broadcast before the instruction is written into the reservation stations. This table is the analogue of the busy-bit table in the MIPS R10000 [5]. For each physical register, an entry in the table stores its DELAY field, MATCH bit, and SHIFT field. When a destination tag is broadcast, the MATCH bit and SHIFT field of the entry corresponding to the tag is updated just like the MATCH bit and SHIFT field of the wakeup logic for a matching source operand.

During decode, an instruction’s execution latency is determined. After the instruction’s register operands have been renamed, the table entry corresponding to the instruction’s physical destination register is updated. Its DELAY field is set to a value derived from the instruction’s execution latency, its MATCH bit is reset, and its SHIFT field is set to 0. Each of the instruction’s physical source registers then accesses the table to determine its DELAY field, MATCH bit, and SHIFT field.

3.3. Select Logic

The select logic for each functional unit grants execution to one ready instruction. If more than one instruction requests execution, heuristics may be used for choosing which instruction receives the grant [1]. The inputs to the select logic are the request signals from each of the functional unit’s RSEs, plus any additional information needed for scheduling heuristics such as priority information. Implementations of the select logic are discussed elsewhere [3] and will not be covered in this paper. As shown in Figure 5, when both READY bits are set, the instruction requests execution. If the instruction receives a grant, its destination tag is broadcast on the tag bus. The execution grant must also be able to clear the MATCH, SHIFT, and READY fields of the RSE so that the instruction does not re-arbitrate for selection.

3.4. Dependent Instruction Execution

In this implementation of conventional scheduling logic, an instruction wakes up in the last half of a clock cycle, and is potentially selected in the first half of the next clock cycle. Note that the wakeup and selection of the instruction straddles a clock edge. If the instruction is selected, the grant from the select logic gates the instruction’s destination tag onto the tag bus, which is then fed to the tag comparators of the wakeup logic. Thus, the tasks of selection, tag broadcast, and wakeup must all occur within in one cycle in order for dependent instructions to wakeup in consecutive cycles.

Figure 7 is an example of the conventional scheduling operation. It shows the pipeline diagram for the execution of the left three instructions of the data flow graph in Figure 2. This schedule assumes each instruction has a 1-cycle latency, and all other parents and grandparents of the ADD are already done. In cycle 1, the READY bit for the XOR’s last source is loaded with a 1. In cycle 2, the XOR is selected for execution. Its destination tag is broadcast on the tag bus, and the MATCH, SHIFT, and READY fields of its RSE are cleared so that it does not request execution again. In this same cycle, the SUB matches the tag broadcast by the XOR, and a 1 is loaded into its READY bit. In cycle 3, the SUB is selected, it broadcasts its destination tag, and wakes up the ADD. . .

Clock:	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
XOR:	Wakeup	Select/ Broadcast	Reg Read	Execute/ Bypass		
SUB:	Wait	Wakeup	Select/ Broadcast	Reg Read	Execute/ Bypass	
ADD:	Wait	Wait	Wakeup	Select/ Broadcast	Reg Read	Execute/ Bypass

Figure 7. Execution of a Dependency Chain Using 1-Cycle Conventional Scheduling

3.5. Pipelined Conventional Scheduling Logic

To break the conventional scheduling logic into a 2-cycle pipeline, a latch must be added in the path of the select logic, tag broadcast, and wakeup logic. We will assume the execution grant from the select logic is latched. Hence the select logic takes one cycle, and the tag broadcast and wakeup logic take one cycle.

Because a latch has been inserted in what was previously an atomic operation, there is a minimum of 2 cycles between the wakeup of dependent instructions. If a parent instruction has a 1-cycle execution latency, this will create a 1-cycle bubble in the execution of the dependency chain. If the parent takes two or more cycles to execute, the bubble can be avoided by using a different encoding for the DELAY field. For 2-stage pipelined scheduling logic, the DELAY field for any 1-cycle *or 2-cycle* operation should be encoded as all 1s. For a latency of N ($N > 1$) cycles, the

Clock:	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7	Cycle 8	Cycle 9
ADD:	Wakeup	Select	Broadcast/ Reg Read	Execute					
LOAD:	Wait	Wait	Wakeup	Select	Broadcast/ Reg Read	Execute	Execute	Execute	
OR:	Wait	Wait	Wait	Wait	Match Tag	Wakeup	Select	Broadcast/ Reg Read	Execute

Figure 8. Execution of a Dependency Chain Using 2-Cycle Pipelined Conventional Scheduling

DELAY field should contain N-2 zeros in the least significant bits, and the upper bits would be set to 1.

Figure 8 shows an example of executing the dependency chain ADD $\rightarrow$ LOAD $\rightarrow$ OR using 2-cycle pipelined scheduling logic. The ADD instruction wakes up in cycle 1 and is selected for execution in cycle 2. The ADD is the parent of the LOAD, and has a 1-cycle latency, so the LOAD's DELAY field is '1111'. In cycle 3, the ADD broadcasts its tag. The LOAD matches, loads its SHIFT field with the contents of the DELAY field, and thus wakes up. The load instruction is selected for execution in cycle 4, and broadcasts its tag in cycle 5. The tag match triggers the OR to set its MATCH bit and load the SHIFT field with the contents of the DELAY field. Since the LOAD is a 3-cycle operation, the contents of the DELAY field for the OR's source would be '1110'. At the end of cycle 6, the value '1111' is shifted into the OR's SHIFT field, and the OR wakes up. In cycle 7, the OR is selected for execution. Note that because the ADD is a 1-cycle operation, there is a 1-cycle bubble between the ADD and the LOAD. However, there is no bubble between the LOAD and the OR since the extra scheduling cycle is hidden by the execution latency of the LOAD.

4. Pipelined Scheduling with Speculative Wakeup

In the last section, we showed that pipelining the conventional scheduling logic introduces pipeline bubbles between dependent instructions. In this section, we show how to pipeline this logic over two cycles without having these bubbles.

Here is a brief overview of this technique: If the parents of an instruction's parent have been selected, then it is likely that the parent will be selected in the following cycle (assuming the parent's parents are 1-cycle operations). Thus, for scheduling logic pipelined over 2 cycles, the child can assume that when the tags of the grandparent pair have been received, the parent is probably being selected for execution and will broadcast its tag in the following cycle. The child can then speculatively wakeup and be selected the cycle after its parent is selected. Because it is not guaranteed that the parent will be selected for execution, the wakeup is only speculative.

Sections 4.1, 4.2, and 4.3 describe implementations of the rename, wakeup, and select logic for this scheduling mechanism. Section 4.4 gives an example of the scheduling operation. Section 4.5.1 discusses an implementation that reduces the amount of logic and state that is kept in the reservation stations. Section 4.5.2 shows that scheduling with speculative wakeup also works for machines that have instruction re-execution.

4.1 Rename Logic

For pipelined scheduling with speculative wakeup, the rename logic is responsible for determining the destination tags of the instruction's grandparents as well as the destination tags of its parents. The grandparents are required for speculative wakeup, which will be described in the next section. At the end of rename, each of the instruction's architectural source register identifiers has been replaced by a parent's destination tag, *and* the set of destination tags of that parent's parents.

To do this, the rename map must be modified. Each map entry is extended so that, in addition to the original physical register identifier, it contains the set of identifiers of the physical registers that are needed to compute the value of the physical register specified by the original identifier. That is, for the instruction that updated the entry, the entry contains the instruction's destination tag, and the set of destination tags of the instruction's parents.

At the beginning of rename, an instruction's architectural source register identifiers are used to index the rename map. Each rename map lookup yields one of the instruction's parent's destination tag, and the set of destination tags of that parent's parents. At the end of rename, the instruction's destination tag is known, and the destination tags of the instruction's parents are known. This information is used to update the map entry whose index is equal to the instruction's architectural destination register identifier.

The dependency analysis logic also needs to be modified. For conventional scheduling, when there is a dependency between two instructions in the group of instructions being renamed, the destination tag of the earlier instruction must be selected as the source tag of the later, dependent instruction. This ensures that the dependent instruction has the correct destination tag for its parent. For pipelined scheduling with speculative wakeup, the logic must also ensure that the dependent instruction has the correct destination tags for that parent's parents. Note that the correct destination tags for that parent's parents are simply the parent's source tags.

To account for this, the set of destination tags for a particular parent's parents is determined by a MUX. An example of one of these MUXes is shown in Figure 9. The MUX in this figure selects the source tags of the third instruction's first parent; i. e., it selects the first two grandparent tags. The inputs to this MUX come from the rename map and the outputs of the MUXes shown in Figure 4. The first input to the MUX is the set of destination tags for the parent's parents that were read from the rename map, and is selected when the parent is not in the group. In Figure 9, this input, labeled Parent_IDs(Op3_src1), comes from the same entry of the rename table as Phy_ID(Op3_src1) shown in Figure 4. The second input to the MUX is the set of source tags for the first instruction in the group, and is selected if the parent is the first instruction in the group. The third input to the MUX is the set of source tags for the second instruction in the group, and is selected if the parent is the second instruction in the group. The control for this MUX is the same as the control used for the MUX to select the third instruction's first source tag shown in Figure 4. Hence, this MUX adds at most one input-to-output delay to the critical path of the rename logic. (The control for the MUX is determined well before the input arrives.)

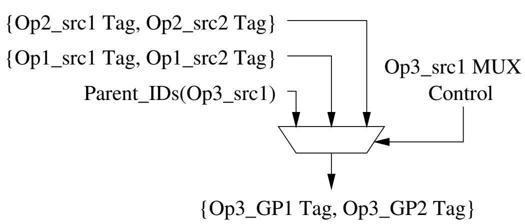


Figure 9. A Grandparent MUX

4.2. Wakeup Logic

The reservation stations contain wakeup logic for parents and grandparents. The wakeup logic for each source is identical to that explained in Section 3.2. As in the conventional 2-cycle pipelined scheduling logic, the DELAY field for each parent contains N-2 zeros in the least significant bits, and ones in the other bits, where N is the execution latency of the parent. The DELAY field for each grandparent contains N-1 zeros in the least significant bits, and ones in the other bits, where N is the latency of the grandparent.

Not all parent and grandparent fields are used for wakeup. If a parent instruction is a multi-cycle operation, then the fields for its parents can be ignored; that is, their RSE fields can be marked invalid. Instead only the tag of the multi-cycle parent will be used. As explained in Section 3.5, there is no pipeline bubble between the execution of a multi-cycle parent and its child for 2-cycle pipelined scheduling logic.

Figure 10 shows how the READY bits for the parent and grandparent tags are used to form the request sent to the select logic. A request is generated when the instruction is speculatively ready. As shown, an instruction is speculatively ready when, for each parent, the parent's parents are ready or the parent is ready.

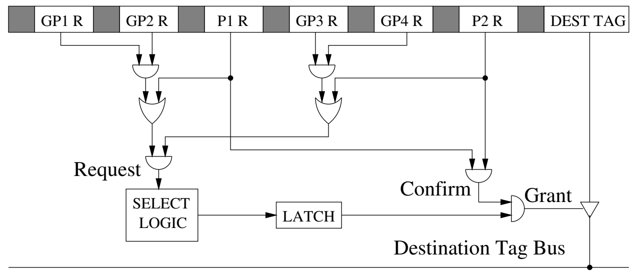


Figure 10. Speculative Wakeup Logic

Although the logic path to the request line shown in this figure has two additional gates compared to the implementation shown in Figure 5, this path could be shortened by taking advantage of the fact that no more than four sources are needed for wakeup. If a parent is a multi-cycle operation, then the RSE field for the parent can be copied into one of the grandparent fields before placing the instruction in the reservation station. In this case, a 4-input AND gate would replace the AND-OR-AND path.

4.3 Select Logic

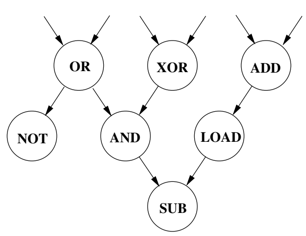
The request line shown in Figure 10 only indicates that the instruction is speculatively ready, so the confirm line is used to verify that the instruction is ready to execute. The confirm line is only asserted when all of the instruction's parents are ready, and is typically asserted one cycle after the request line. If the request has the highest priority, and the confirm line is asserted, which means the instruction is really (non-speculatively) ready, the request is granted. Otherwise, the request is not granted.

False selection occurs whenever a request is not confirmed. That is, if the request line is asserted, and the confirm line is not asserted one cycle later, false selection occurs. False selection is a performance problem when it prevents a *really* ready instruction from being selected. This only occurs when the request that wasn't confirmed had the highest priority and there were really ready instructions that could have been selected.

For pipelined scheduling with speculative wakeup, an instruction and its parents can assert their request lines at the same time. The machine must guarantee that the parents are eventually selected for execution, otherwise, deadlock can occur. There are many ways to provide this guarantee. The most obvious is to use instruction age to assign selection priorities; i. e., older instructions (instructions that occur earlier in the dynamic instruction stream) have priority over younger instructions. Another is to use a round robin scheme to assign selection priorities.

4.4. Dependent Instruction Execution

To illustrate the operation of pipelined scheduling with speculative wakeup, consider the dependency graph in Figure 11a. All instructions are 1-cycle operations except for the LOAD, which takes 3 cycles. The SUB instruction will wakeup on the tag broadcasts of the OR, XOR, and LOAD instructions since the AND is a 1-cycle operation and the LOAD is a 3-cycle operation. The DELAY field for the SUB's first parent, the AND, will contain the value '1111'. This field will only be used for confirmation after selection has occurred. The DELAY fields for the SUB's first two grandparents, the OR and XOR, will contain the value '1111'. The DELAY field for the SUB's second parent will be encoded as '1110' so that the SUB will delay its wakeup for at least 1 cycle after the tag of the LOAD is broadcast. The last two grandparent fields for the SUB are marked as invalid, since they will not be used.



(a)

	Clock:	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6
RS_1	OR:	Select	Broadcast/ Reg Read	Execute			
	NOT:	Wakeup	Select	Broadcast/ Reg Read	Execute		
	AND:	Wakeup	Select (no grant)	Select	Broadcast/ Reg Read	Execute	
RS_2	SUB:	Wait	Wakeup	(false) Select	Select	Broadcast/ Reg Read	Execute

(b)

Figure 11. Example of Speculative Wakeup

Figure 11b shows an example of the scheduling operation of this data flow graph when false selection occurs. For this example, assume that the XOR, ADD, and LOAD instructions have already executed by cycle 1. Also assume that all instructions except the SUB are in the same set of reservation stations (designated by RS_1), and the SUB is in a second set (RS_2). (There are no extra delays for broadcasting tags or data between sets of reservation stations in this example.) In cycle 1, the OR is selected for execution and the AND and NOT wakeup on the tag broadcast by the OR's last parent. In cycle 2, the SUB wakes up after matching the tag broadcast by the OR. The AND and NOT both request execution in this cycle, but only the NOT receives an execution grant. The NOT and AND also match the tag broadcast by the OR instruction. In cycle 3, the AND and SUB are both selected for execution. In cycle 4, the AND's selection is confirmed and its tag is broadcast, but the SUB's selection is not confirmed. This false selection may have prevented the selection of another instruction in RS_2 in cycle 3. The SUB requests execution again, and the selection is confirmed in cycle 5.

4.5. Implementation Considerations

4.5.1. Reducing the Cost of the Wakeup Logic. We assume the tag buses running through each RSE set the size of the reservation stations, and that the wakeup logic associated with each RSE can easily be hidden underneath all these wires. If that is not the case, it is possible to reduce the number of tags—and thus, size of the wakeup logic—required to perform pipelined scheduling with speculative wakeup. Two observations are required to understand how this can be done.

The first: An instruction can always wake up using its parent tags. The grandparent tags are only provided for performance reasons. If some (or all) of them are missing, the machine still functions correctly.

The second: An instruction becomes ready for execution only after all its parents' destination tags have been broadcast. If the machine can predict which parent will finish last, it can use only the destination tags of that parent's parents to become speculatively ready. That is, each RSE will have the tags of all the instruction's parents, and, for the parent that was predicted to be last, the tags of its parents. If the prediction is wrong, the machine still functions correctly: the instruction just isn't selected as soon as it could be.

We used a simple scheme to predict the last parent. A 2-bit saturating counter was stored along with each instruction in the instruction cache. When an instruction was fetched, the upper bit of the counter specified whether the first or second parent would finish last. During rename, only the parents of the parent that was predicted to finish last were stored in the grandparent fields. While an instruction was in the reservation stations, it recorded which parent finished last. [3] When the instruction committed, it decremented the

[3] Actually, the only time the grandparent fields are used is when the parent has a latency of 1 cycle. The instruction actually recorded which parent with a 1 cycle latency finished last.

counter if the first parent finished last, and incremented it if the second parent finished last.

4.5.2. Operation of Instruction Re-execution. The scheduling logic presented can handle instruction re-execution without any modification. As mentioned in Section 2.1, the child of a load instruction is scheduled assuming the load hits the cache. If the load misses, or if the load is found to have a memory dependency on an earlier store instruction, the load result may be delayed. When this happens, the chain of instructions dependent on the load must be re-scheduled. This is accomplished by rebroadcasting the load's destination tag on the tag bus. When the tag is rebroadcast, all children of the load will rewake since loads are multi-cycle operations. All grandchildren of the load will either rewake immediately, or if the dependent parent is a multi-cycle operation, will rewake after that parent broadcasts its tag. Hence there is never a situation where a dependent of the load is not rewakened.

5. Experiments

To measure the impact of pipelining the scheduling logic, we modeled four machines: a *baseline* machine, which uses conventional scheduling pipelined over 2 cycles; a *budget* and a *deluxe* machine, which use 2-cycle pipelined scheduling with speculative wakeup; and an *ideal* machine, which uses conventional 1-cycle scheduling logic. The budget machine uses RSEs that can hold only two grandparent tags. Last parent prediction is used to select which two. The deluxe machine uses RSEs that can hold all grandparent tags. As mentioned in Section 3.5, the baseline machine only introduces pipeline bubbles when scheduling single-cycle instructions, not when scheduling multi-cycle instructions.

All machines were 8-wide superscalar processors with out-of-order execution, configured as shown in Table 1. They required 2 cycles for fetch, 2 for decode, 2 for rename, 1 for register read, and 1 for commit. The ideal machine required 1 cycle for wakeup/select. The others required 2. The execution latencies are shown in Table 2. An instruction with a 1-cycle execution latency requires a minimum of 10 cycles to progress from the first fetch stage to commit.

The machines were simulated using a cycle-accurate, execution-driven simulator for the Alpha ISA. Figure 12 shows the IPC of the four machines over the SPECint95 benchmarks. The 15% IPC difference between the baseline and ideal machines represents the amount of performance that can be gained by using pipelined scheduling with speculative wakeup. The deluxe machine gains 86% of this difference. On average, the deluxe machine performs 13% better than the baseline machine, and within 2% of the ideal machine. The budget machine performs within 1% of the deluxe machine.

Branch Predictor	15-bit gshare, 2048-entry BTB
Instruction Cache	64KB 4-way set associative (pipelined) 2-cycle directory and data store access
Instruction Window	128 RSE (16 for each functional unit)
Execution Width	8 multi-purpose functional units, only four of which support load/store operations
Data Cache	64KB 2-way set associative (pipelined) 2-cycle directory and data store access
Unified L2 Cache	1MB, 8-way, 7-cycle access 2 banks, contention is modeled

Table 1. Machine Configuration

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

Table 2. Instruction Class Latencies

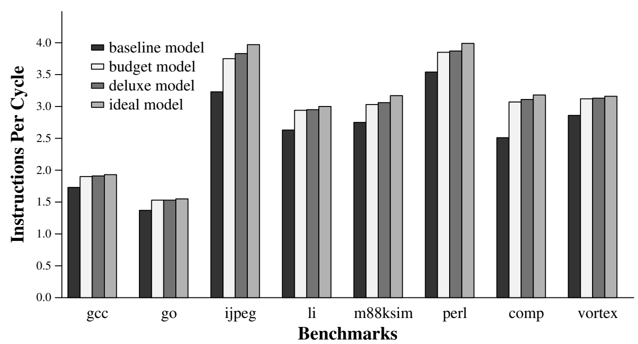


Figure 12. Comparison of the Four Machines

The IPC differences between the budget, deluxe, and ideal machines are primarily due to false selections caused by speculative wakeup. False selections only impact IPC if other instructions that were ready to execute were prevented from executing due to a false selection. Figure 13 shows the amount of false selections and scheduling opportunities lost due to false selections for the budget and deluxe machines. This graph shows the fraction of scheduling opportunities in which a selection resulted in: (1) a false selection that prevented a ready instruction from receiving an execution grant, (2) a false selection, but no instructions were ready, and (3) a correct selection. The fourth case—the only other possible case—is when no instructions request execution. The first of each pair of bars (bars with solid colors) show the cycle breakdown for the budget machine. The second of each pair (striped bars) show the breakdown for the deluxe machine. As the graph shows, the selection logic for a given

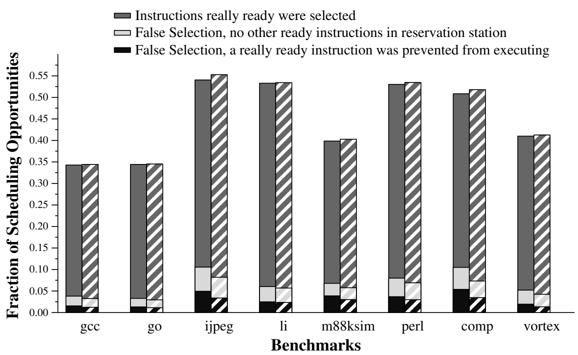


Figure 13. Selection Outcomes

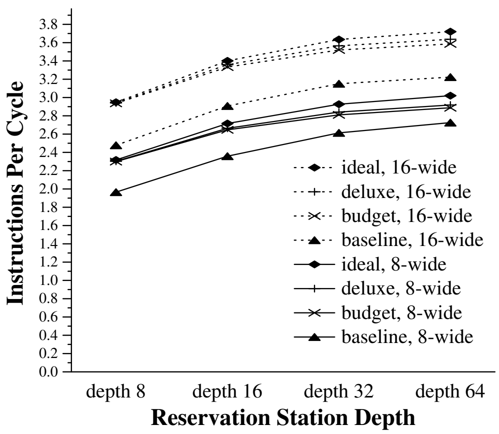


Figure 14. Comparison of the Four Machines for Different Window Sizes

functional unit awarded an execution grant 35% to 55% of the time. A ready instruction was prevented from executing due to a false selection only 1% to 7% of the time. Note that these metrics cannot be compared to IPC since they measure both non-speculative *and speculative* instructions.

The frequency of false selections is quite sensitive to the amount of functional unit contention. Simulations of the four machines using both 8 and 16 functional units and several instruction window sizes were run to demonstrate the effect of this contention on IPC. The simulation parameters of the 16-wide issue machines are similar to those of the 8-wide machines except that 8 of the 16 functional units are capable of executing load and store instructions. Figure 14 shows the harmonic mean of the IPC of the SPECint95 benchmarks for all models using both issue widths when the number of RSEs for *each* functional unit is varied from 8 to 64. The machine model with the most contention for execution resources is the 8-wide machine with a 512-entry instruction window. For this configuration, pipelined scheduling with speculative wakeup gains 65% of the difference between the baseline and ideal machines.

6. Conclusion

This paper demonstrates that the dynamic instruction scheduling logic can be pipelined without sacrificing the ability to execute dependent instructions in consecutive cycles. It introduced pipelined scheduling with speculative wakeup, which pipelines this logic over 2 cycles. This technique significantly reduces the critical path through this logic while having only a minor impact on IPC. If the critical path through this logic limits the processor cycle time, this technique allows microarchitects to build higher performance machines by enabling higher clock frequencies, deeper pipelines, and larger instruction windows.

Acknowledgements

We especially thank Paul Racunas, Jean-Loup Baer, and the anonymous referees for their comments on earlier drafts of this paper. This work was supported in part by Intel, HAL, and IBM. Mary Brown is supported by an IBM Cooperative Graduate Fellowship.

References

[1] M. Butler and Y. Patt, "An investigation of the performance of various dynamic scheduling techniques," in *Proceedings of the 25th Annual ACM/IEEE International Symposium on Microarchitecture*, 1992.

[2] *IA-32 Intel Architecture Software Developer's Manual With Preliminary Willamette Architecture Information Volume 1: Basic Architecture*, Intel Corporation, 2000.

[3] 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.

[4] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[5] K. C. Yeager, "The MIPS R10000 superscalar microprocessor," *IEEE Micro*, vol. 16, no. 2, pp. 28–41, April 1996.

[6] A. Yu, *Client Architecture for the New Millennium*, Intel Corporation, February 2000. Spring 2000 Intel Developer Forum Keynote Presentation.