Understanding Scheduling Replay Schemes

Ilhyun Kim and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
{ikim,mikko}@ece.wisc.edu

Abstract

Modern microprocessors adopt speculative scheduling techniques where instructions are scheduled several clock cycles before they actually execute. Due to this scheduling delay, scheduling misses should be recovered across the multiple levels of dependence chains in order to prevent further unnecessary execution. We explore the design space of various scheduling replay schemes that prevent the propagation of scheduling misses, and find that current and proposed replay schemes do not scale well and require instructions to execute in correct data dependence order, since they track dependences among instructions within the instruction window as a part of the scheduling or execution process.

In this paper, we propose token-based selective replay that moves the dependence information propagation loop out of the scheduler, enabling lower complexity in the scheduling logic and support for data-speculation techniques at the expense of marginal IPC degradation compared to an ideal selective replay scheme.

1. Introduction & Motivation

Over the last 15 years, microprocessors have evolved from relatively straightforward, pipelined, but largely non-speculative implementations like the MIPS R2000, to highly speculative, deeply pipelined machines that execute instructions out of order and speculate past numerous serializing conditions to reap maximum performance benefit. Current generation designs already speculate on load latency by assuming cache hits, predict that loads will hit in-flight stores, hoist loads speculatively to overcome multiprocessor ordering constraints, and, of course, predict conditional branches and fetch speculative instructions from the predicted path. Looking ahead, we can reasonably expect that the continuing demand for higher clock frequency will push designers to deeper pipelines and a greater mismatch between core frequency and memory system latencies. Continuing to deliver high performance will require even more aggressive application of such speculative techniques, as well as possible new ones like value prediction [11], memory cloaking [13], or other techniques yet to be invented.

All speculative techniques share a few common requirements: there must be some mechanisms for generating predictions, there must be microarchitectural support for realizing the benefit of those predictions, and there must be recovery mechanisms in place that guarantee correctness in those cases where speculation was incorrect. Predictors and speculative techniques have been studied at length in the literature since any compelling presentation of a new idea must necessarily demonstrate its benefits. However, relatively little focus has been placed on effective recovery mechanisms, short of vague descriptions of recovery schemes like *refetch*, *squash, reissue*, and *replay.* Only in the realm of branch prediction, where recovery consists of simply squashing all microarchitectural state beyond the mispredicted branch, and refetching all subsequent instructions, recovery techniques have been satisfactorily described. In the realm of speculative scheduling and data speculation techniques, there is very little in the way of satisfactory descriptions of the specifics of the recovery mechanisms. This paper addresses this deficiency in the literature with a systematic treatment of recovery techniques and presents a framework for analyzing their complexity. We conclude that universal selective replay, where an instruction can cause a recovery event at any point during its lifetime, is barely feasible for current-generation designs, and does not scale to wider machines or additional types of speculation. In response, this paper proposes a new, targeted replay scheme that uses prediction to identify likely causes of misprediction events within a processor, and focuses demonstrably expensive replay resources on those instructions. We show that our proposed scheme enables lower complexity in the scheduling logic and support for data speculation techniques, with slightly reduced performance over an ideal replay scheme.

The remainder of the paper is structured as follows. Section 2 describes the problem with incorrect propagation of the speculative execution wavefront. Section 3 explores the design space of scheduling replay schemes, and identifies potential limitations. Section 4 proposes token-based selective replay. Section 5 provides detailed performance evaluation of various scheduling replay schemes.

2. Background

2.1. Initiation, Propagation and Termination of Speculative Execution Wavefront

Out-of-order processors are based on Tomasulo's algorithm [7], in which instructions that finish execution wake up their dependent instructions and the scheduling logic selects issue candidates from the pool of ready instructions. For back-to-back dependent operation execution, this wakeup and select process must occur atomically (i.e. it cannot be easily pipelined [5][6]) in parallel with execution. In recent microprocessor designs, the number of pipeline stages between instruction issue and execution has

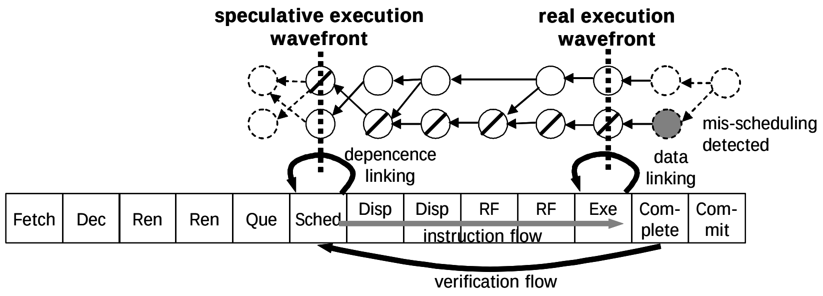


Figure 1. Base machine pipeline and wavefront propagation.

increased to accommodate the latency needed for reading the register file and performing other book-keeping duties. As the instruction issue and execution stages are further separated, a naive implementation would fail to achieve maximum ILP because back-to-back execution of dependent instructions is no longer possible. To address this problem, current-generation processor implementations [2][3] use *speculative scheduling* in which the scheduler speculatively wakes up and selects dependent instructions several clock cycles before the actual execution.

This process is illustrated in Figure 1. In the schedule stage, the wakeup and select operations link data dependences among instructions, and initiate the *speculative execution wavefront* [21]. This speculative image of execution is projected to the execution stage and drives the actual instruction execution, creating the *real execution wavefront*. However, dynamic events (e.g. cache misses or memory dependences) can cause latencies predicted at schedule time to be incorrect, and the actual instruction execution may diverge from the scheduled execution. In this *scheduling miss* case, the current incorrect execution should be canceled and the instructions should be rescheduled. If the schedule is correct, corresponding instructions should also be notified so that the processor's critical resources, i.e. the issue queue entries, can be reclaimed, or a branch misprediction can be resolved earlier without waiting to reach commit. Therefore, actual instruction execution serves to verify the speculative schedule.

Although the detection of a scheduling miss is straightforward (i.e. comparing the predicted and actual execution latencies), efficiently reclaiming instructions from the incorrect schedule is complicated by the delay between the speculative and real execution wavefronts. Naively, the verification status can propagate serially, along the data dependence edges, as instructions are executed. This *serial verification* that triggers rescheduling of directly dependent instructions can be implemented just like broadcast-based wakeup. However, without the aid of structural hazards such as window fills or early data dependence chain terminations by mispredicted branches, verification may not catch up with the speculative execution wavefront. This is illustrated in Figure 2a, and occurs because both the speculative execution wavefront and the verification propagate through data dependence chains at the same rate of one level per cycle. In this serial verification example, a scoreboard is integrated into the register file and the bypass network to propagate verification status to dependent instructions, and communicates with the instruction scheduler by broadcasting output register identifiers of mis-scheduled instructions. Figure 3 shows the distribution of load scheduling misses as a function of how far the speculative execution wavefront initiated by scheduling misses is propagated on an 8-wide machine. We observe that load scheduling misses are propagated through 836 instruction levels[1] in the worst case (*parser*), which increases the total issue count by 15% (on average 9.9%, worst 42.1% in *mcf*), compared with a parallel verification scheme that will be explained shortly. The wasted issue bandwidth may negatively affect performance and consume excess power.

To avoid this problem, speculative scheduling needs *parallel verification* in which instructions in the entire dependence chain are verified in parallel, as shown in Figure 2b, so that the mis-scheduled execution wavefront does not trigger any further instruction issue.

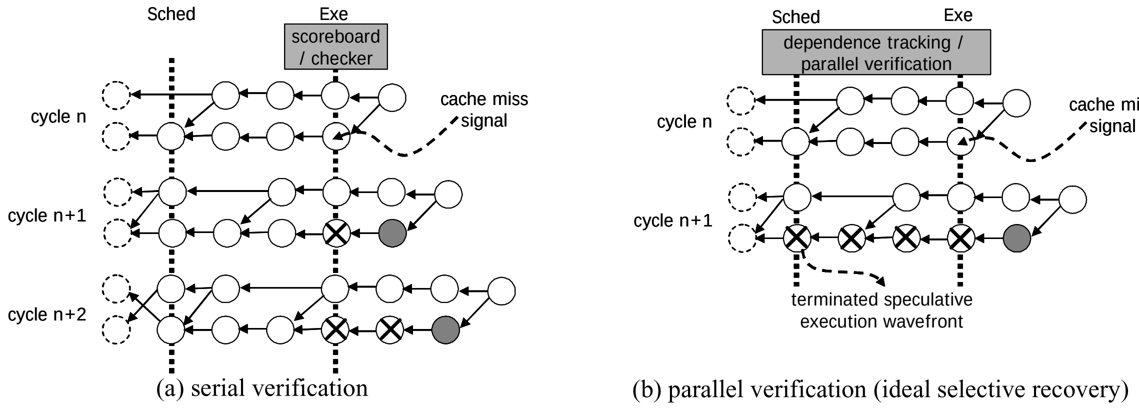

Sched
Exe
scoreboard / checker
cache miss signal
cycle n
cycle n+1
cycle n+2
(a) serial verification
Sched
Exe
dependence tracking / parallel verification
cache miss signal
cycle n
cycle n+1
terminated speculative execution wavefront
(b) parallel verification (ideal selective recovery)

Figure 2. The speculative execution wavefront in serial and parallel verification.

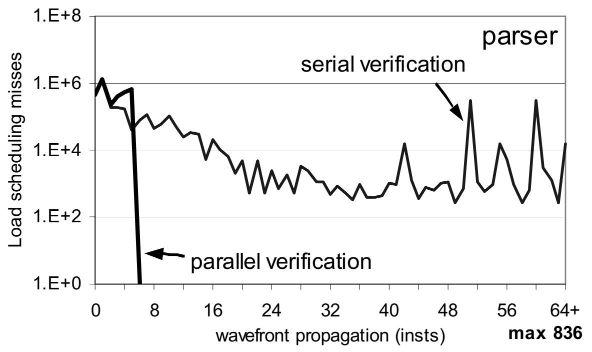


Figure 3. Speculative wavefront propagation in serial and parallel verification.

2.2. Requirements of Parallel Verification

Scheduling replay that performs parallel verification requires the following necessary conditions.

- The propagation of scheduling verification must be faster than that of the speculative execution wavefront so that recovery eventually catches up with the invalid speculative execution wavefront.
- Scheduling replay must be performed on the transitive closure of dependent instructions. If it does not cover the entire region, some portion of the speculative execution wavefront that slips through invalidation may not be terminated.

Based on these conditions, ideal scheduling replay is achieved if all mis-scheduled dependent instructions are invalidated instantly and independent instructions are unaffected by scheduling miss events. Due to the delay between speculative and real execution wavefronts, determining if a mis-scheduling event is relevant to the instructions currently being scheduled requires multiple levels of data dependence tracking. In the machine model assumed in this paper (and many current generation implementations), scheduling misses can occur at load instructions due to access latency misprediction (e.g. cache misses), and store-to-load aliasing when the store data is not ready. Therefore, an instruction should keep track of all possible combinations of direct or indirect parent loads that the instruction depends on. This functionality can be naively achieved by storing a dependence vector with every instruction, in which the number of bits is equal to the maximum number of loads in the window. Similarly, if future-generation microprocessors require scheduling replay at any instruction boundary, the size of the dependence vector may become as big as the number of in-flight instructions because an instruction may depend on all other instructions in the window. However, this would be impractical due to hardware complexity since the number of wires to signal verification status to the scheduler is proportional to the dependence vector size.

1. Note that the speculative execution wavefront propagation is not bounded by the size of the instruction window (128 instructions in this experiment) because the propagation can be sustained through newly inserted instructions into the window as older instructions that are successfully replayed release their issue queue entries.

Propagation distance	The number of load ports					
	1	2	4	8	16	32
1	1	2	2	2	2	2
2	2	3	4	4	4	4
3	3	4	5	8	8	8
4	4	6	8	12	16	16
5	5	8	12	16	24	32
6	6	10	16	24	32	48
7	7	12	20	32	48	80

Table 1: The maximum number of parent loads for an instruction to track for scheduling replay.

2.3. Dependence Tracking

Since issue bandwidth limits the propagation of the speculative wavefront, in fact, the actual number of parent loads to track is usually smaller than the maximum number of in-flight loads in the machine, and becomes mostly a function of the number of possible misses per cycle (i.e. the number of load ports) and the pipeline depth.

Table 1 shows the maximum number of loads for an instruction to track in various pipeline configurations. Since the general equation derived from a graph model is complex, we assume the following for the table: 1) only loads incur scheduling misses, 2) the load latency is greater than 1 clock cycle, 3) an instruction has up to two source operands, 4) the load issue bandwidth is no greater than the issue bandwidth for single-cycle instructions.

The *propagation distance* is defined as the sum of the distance from schedule to execute stages and the verification latency. For example, the pipeline model shown in Figure 1 has the propagation distance of 6 since the pipeline distance from schedule to execution is 5, and the verification latency is 1, assuming that a miss is signaled from the completion stage. Note that the propagation distance is independent of load latency since the scheduler will not issue dependent instructions for the period in which the speculative execution wavefront does not propagate. The two shaded boxes are the entries corresponding to our 4 and 8-wide machine configurations. From the table, we find that the number of loads to track is small relative to the window size. However, these numbers do not imply the number of bits in a dependence vector but rather the degree of concurrency. All loads in the window should still be mapped to unique names so that dependent instructions can track the validity of multiple parent load instructions.

Therefore, efficient scheduling replay implementations need to approximate or convert the name space for precise dependence tracking into a smaller set with a manageable number of bits. When dependence tracking is approximated, there is a trade-off between *precision* of replay and hardware cost. In general, hardware cost decreases as precision drops and the replay region grows. At the same time, the scheduling miss penalty increases as more independent instructions are unnecessarily replayed.

In the next section, several possible implementations of scheduling replay schemes will be discussed. Section 4 will identify the potential difficulties in such schemes, and propose a solution that overcomes these limitations.

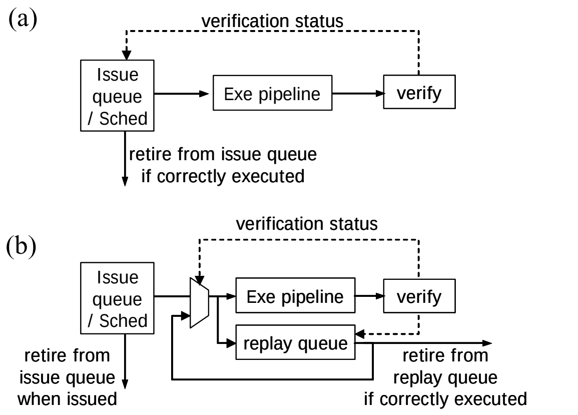


Figure 4. Scheduling replay models. (a) issue-queue-based replay, (b) replay-queue-based replay.

3. Scheduling Replay Implementations

In this section, we describe possible implementations of scheduling replay schemes that perform parallel verification. This section categorizes the scheduling replay schemes by the precision of the replay region.

3.1. Scheduling Replay Models

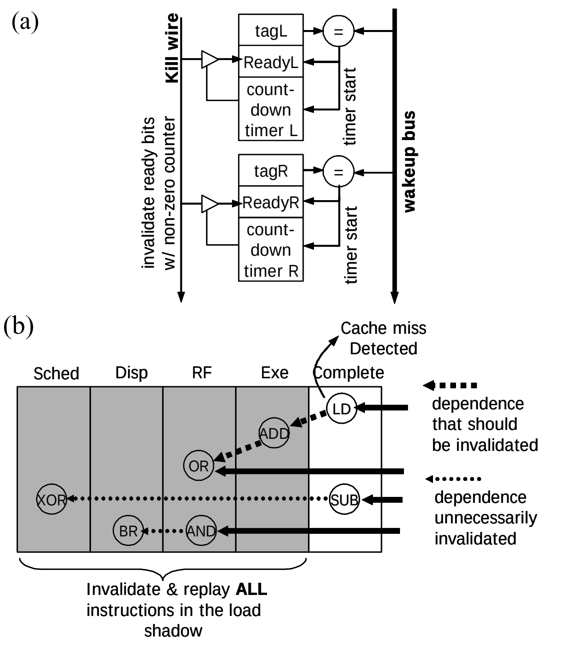
Figure 4 illustrates two possible scheduling replay models. The major difference between them stems from where an issued instruction waits for the outcome of scheduling verification before it completes. The issue-queue-based replay model in Figure 4a is similar to the one that is used in the Alpha 21264 [2], where an instruction can leave the issue queue only after it is verified to be correctly executed. The shortcoming of this approach is that many issue queue entries can be unnecessarily held by correctly scheduled instructions, which reduces the effective size of the window. However, issued instructions can monitor all wakeup and rescheduling activities, which enables instructions to be dynamically scheduled after replay events are detected.

On the other hand, the replay-queue-based model in Figure 4b, which is proposed in [20], allows instructions to leave the issue queue as soon as they are issued, and performs replay operations from a separate queue. Although the issue queue capacity is not negatively affected in this approach, instructions would not be able to dynamically react to replay events once they leave the scheduler, which may incur multiple replays for the same set of instructions.

In either case, scheduling replay requires a mechanism that terminates the incorrect speculative wavefront in order to prevent the scheduler from issuing unnecessary instructions in the presence of rescheduling events. In this paper, we assume that the base machine has the issue-queue-based replay model for various scheduling replay schemes, and leave detailed exploration of the hardware design space of the replay-queue-based model to future work.

3.2. Refetching Replay

The branch misprediction recovery mechanism can be used for scheduling replay. A scheduling miss is treated as a branch misprediction: all instructions that follow the



(a)
Kill wire
invalidate ready bits w/ non-zero counter
tagL
ReadyL
count-down timer L
tagR
ReadyR
count-down timer R
=
timer start
wakeup bus
(b)
Cache miss Detected
Sched
Disp
RF
Exe
Complete
LD
ADD
OR
XOR
SUB
BR
AND
dependence that should be invalidated
dependence unnecessarily invalidated
Invalidate & replay ALL instructions in the load shadow

Figure 5. Non-selective, wakeup order-based replay.

misprediction in program order are flushed out of the processor, and instruction fetch is redirected to refetch those instructions. Although this replay can be easily integrated into existing hardware, the replay region is most imprecise and it has the obvious drawback that the mis-scheduling penalty is quite severe.

3.3. Non-Selective Replay

This scheme is better known as squashing replay and is used in the Alpha 21264 [2], in which all dependent and independent instructions issued after the load scheduling miss are invalidated and replayed.

Figure 5a illustrates a possible implementation of a non-selective replay mechanism. Conceptually, non-selective replay can be implemented simply by flushing the pipeline between the schedule and execute pipeline stages. In order to keep track of the readiness of source operands, all wakeup activity is timestamped to check if an operand has become ready after the mis-scheduled instruction was issued, implying that the operand may depend on the scheduling miss. The timestamp can be implemented with a countdown timer decremented every clock cycle, which starts counting when the operand is awakened. The initial value of the timer is set to the propagation distance defined in Section 2.3, and therefore the parent instruction completes when the timer becomes zero. If a mis-scheduling event is signaled through the *kill wire* but the timer value is not zero, it is locally determined that the current scheduling miss could be a direct or indirect parent instruction. The ready state of the operand is reset, setting the instruction to the waiting state before the instruction is replayed.

Figure 5b shows a possible scenario of non-selective replay. We assume that the load execution latency is 1 clock cycle for simplicity of presentation. In this scenario, a cache miss is detected for *LD* in the completion stage, and the dependent *ADD* and *OR* should be replayed. These

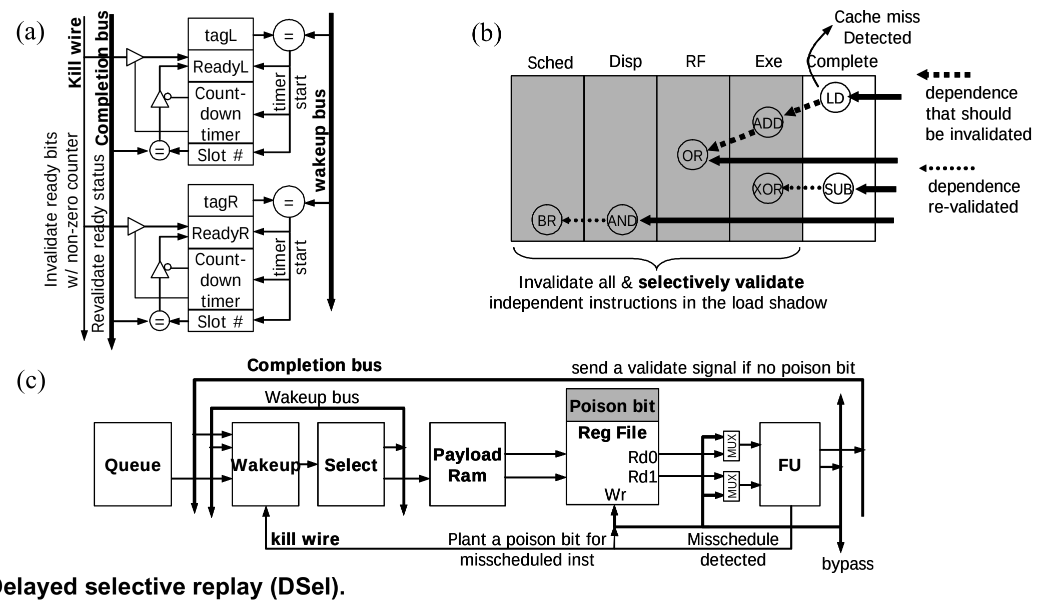


Figure 6. Delayed selective replay (DSel).

dependent instructions have non-zero timer values for dependence edges (shown in dotted lines) from *LD*, and they are invalidated. Once *LD* is issued again when the cache miss is resolved, it will wake up dependent instructions (*ADD* and *OR*) again, triggering replay. The source operand of the *AND* is not invalidated because the parent instruction has retired and the timer value is already zero. Although replaying this instruction is unnecessary, it is also flushed and re-issued to ensure that it once again wakes up its dependent instructions (*BR* in the example) that were invalidated due to a non-zero timer value.

Special handling is required for the unnecessarily invalidated *XOR* since no parent instruction will trigger re-issuing it after recovery. There are several solutions to this problem. One is to use a *completion bus* that broadcasts nonspeculative completion status to dependent instructions (similar to the wakeup bus). A simpler solution is to replay all instructions in the same completion group (*LD* and *SUB* in this example) so that all invalidated operands in the shadow are guaranteed to be awakened again.

The disadvantage of this scheduling replay scheme is its performance scalability. Since this scheme uses approximated time information for dependence tracking, the replay region is not precise and it may negatively affect performance and issue count.

3.4. Selective Replay

Selective replay reschedules only instructions dependent on mis-scheduled loads. The scheduling miss penalty is reduced over non-selective replay but complexity may significantly increase for precise dependence tracking. This section discusses three possible implementations.

3.4.1. ID-based Selective Replay

The most straightforward and precise implementation is to use the load scheduler entry number or physical register identifiers as tags for kill bus broadcast. In this configuration, each instruction has a bit vector with as many bits as the maximum number of load instructions in the window, and the broadcast on the kill bus invalidates any instructions with the matching bits in the vector. Propagating the list of parent instructions can be performed in the rename stage, where the fields for the list of parent instructions are piggy-backed in all rename table entries. Alternatively, this process can also be performed within the issue logic, as discussed in [19]. Although this approach can be made to work in a small-scale pipeline, it would not be feasible in a large-scale pipeline due to the large name space required for all possible combinations of parent loads.

3.4.2. Delayed Selective Replay

The non-selective replay mechanism presented in Section 3.3 can be extended to replay selectively by adopting a completion bus and poison bits. Figure 6 illustrates the changes in issue logic and pipeline. Unlike non-selective replay that flushes all instructions within the execution pipeline, this scheme lets issued instructions flow in the pipeline and then selectively revalidates independent instructions using completion status.

When a scheduling miss is detected, this event is signaled through a kill wire to the scheduler and any operand with a non-zero counter value is invalidated to prevent further propagation of the speculative execution wavefront (same as non-selective recovery). At the same time, a poison bit is planted into the output register of the mis-scheduled instruction (*LD* in the example) through the bypass network and register file, indicating that the result value is invalid. Poison bits are propagated down along dependence edges as a part of execution process when dependent instructions (*ADD* and *OR*) consume invalid source values originating from the mis-scheduled load.

When an instruction reaches the completion stage but does not generate an output value with a poison bit, indicating that it is independent of the scheduling miss, this is broadcast through the completion bus and revalidates the ready status that has been previously invalidated by the kill signal. The functionality of the completion bus can be naively achieved by broadcasting target register identifier

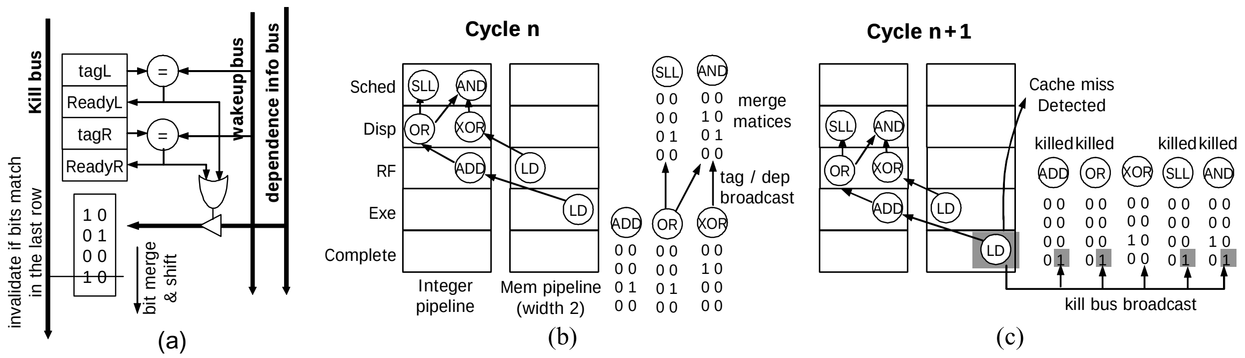


Figure 7. Position-based selective replay (PosSel).

bits, just as the wakeup bus does. Alternatively, a single wire from each functional unit may replace full register identifiers if source operands tracks the issue slots (*Slot #*) in which their parent instructions have been issued, as shown in Figure 6a. Since the countdown timer is initially set to the value that becomes zero when the parent instruction completes, sampling the signal in the given issue slot at timer value zero determines if the parent instruction has successfully completed.

Since dependences are tracked after a mis-scheduling event is detected, this scheme is not as effective as ideal selective replay and incurs unnecessary delays. For example, *BR* in Figure 6b cannot be issued until after its parent instruction (*AND*) reaches the completion stage, which creates at least 3 clock cycles of bubbles between two instructions. This penalty applies to all independent instructions that issue in the shadow of a mis-scheduled load.

3.4.3. Position-based Selective Replay

This is an ideal selective replay scheme that precisely invalidates only dependent instructions without affecting other independent instructions. Unlike ID-based selective replay, the number of tag bits for precise dependence tracking are reduced by using a position-based name space in which bits in a matrix represent the 2-dimensional position of direct and indirect parent loads in the pipeline. As shown in Figure 7a, each issue queue entry has a dependence matrix, whose dimension is determined by the issue width of load instructions (memory pipeline width) and the propagation distance. This matrix is propagated through data dependence edges along with wakeup operations.

For simplicity of presentation, we assume that the load execution latency is 1. In Figure 7b, four rows of the matrix represent the pipe stages from dispatch to completion, and two columns represent issue slots of the memory pipeline. The bits in the matrix are generated by the load instruction, which marks its issue slot in the first row before broadcast. The bits in matrices are shifted down every clock cycle to track instruction flow in the pipeline. The matrices of parent instructions propagate to dependent instructions along with tag broadcast. If a dependent instruction receives two matrices for source operands, (*AND* in the figure), they are merged into one before it is sent out to dependent instructions again. At any given time, all ready operands in the window have the complete list of direct and indirect parent loads in the window.

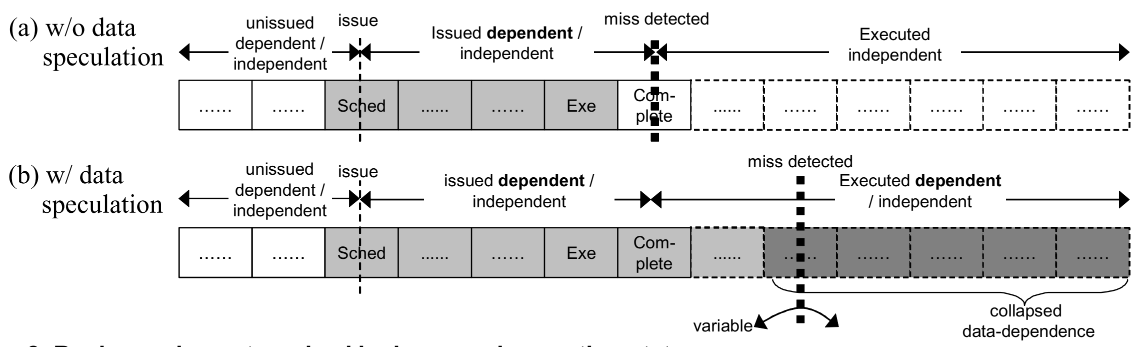
Scheduling misses are detected in the completion stage and the bits in the bottom row of the matrix are the ones that correspond to those misses. Therefore, instructions in the scheduler monitor the kill bus for bits in the bottom row only. The operand is invalidated and the instruction will replay if the current scheduling miss originates from the same column in the bottom row, as shown in Figure 7c.

3.5. Limitations of Scheduling Replay Schemes

Scalability. As the pipeline becomes wider and deeper, non-selective and delayed-selective replay schemes may experience more performance degradation since the region of invalidated instructions under a scheduling miss also grows. Although position-based selective replay does not have this property and can minimize the mis-scheduling penalty, its hardware complexity does not scale well to a wider machine. The number of bits in the dependence matrix is a function of the number of memory ports, the machine issue width and the pipeline depth, and hence the number of wires to propagate dependence matrices may increase quadratically if the machine width and the number of memory ports grow at the same rate. Given the machine configurations used in this paper, the number of wires in the dependence information bus increases from 48 to 192 when transitioning from 4 to 8-wide machines.

Data-dependence Enforcement. As we discussed in Section 2.3, efficient scheduling replay implementations require approximation or conversion of the name space for all possible scheduling misses into a smaller set with a manageable number of bits; non-selective replay schemes use approximated time information; delayed and position-based selective replay schemes rely on time and position. The fundamental assumptions to enable such approximation or conversion are 1) instructions are executed in the correct dependence order that the program defines, and 2) the verification status is known in a fixed number of cycles after issue. The first assumption facilitates correlating data dependences with time (i.e. issue order). The second assumption enables the following: 1) all previously issued instructions can be verified before the unverified instruction reaches the completion stage, which prevents piling up of unverified instructions; 2) wires and structures to propagate verification status are accessible at a single location; 3) dependent instructions can collect parents' verification status implicitly by waiting for counter "time-out".

Figure 8 illustrates the pipeline stages in which dependent instructions may exist when a scheduling or data-



(a) w/o data speculation
unissued dependent / independent
issue
Issued **dependent** / independent
miss detected
Executed independent
......
Sched
Exe
Com-plete
(b) w/ data speculation
unissued dependent / independent
issue
issued **dependent** / independent
miss detected
Executed **dependent** / independent
variable
collapsed data-dependence

Figure 8. Replay region categorized by issue and execution status.

speculation miss is detected. In the figure, extra pipeline stages are conceptually added after the completion stage (presented as dotted lines) to show dependence and time relationship among instructions. Pipeline stages that may contain instructions dependent on the miss are shown as shaded boxes. In Figure 8a (without data speculation) where these assumptions are valid, no dependent instruction could have been issued earlier than the mis-scheduled instruction, and mis-scheduling is always discovered in the completion stage. Therefore, tracking issue and execution status essentially filters out independent instructions because all issued and dependent instructions can be found only within the execution pipeline.

Many researchers found that selective replay is important in many data-speculation techniques [10][11][13][14][22] to realize their compelling benefits. They benefit from collapsing true data dependences and enabling dependent instructions to execute before their source dependences are satisfied. At the same time, they make the propagation distance non-deterministic due to variable verification delay (i.e. latencies for cache miss resolutions are added to the verification delay for load value prediction [10]). Hence, none of the assumptions for scheduling replay schemes holds in such environments, and instructions dependent on the data speculation misses should be searched across all in-flight instructions, as shown in Figure 8b. Although the purpose of replay for both speculative scheduling and data-speculation is similar, the replay schemes discussed in the previous sections, with the exception of ID-based selective replay, cannot be directly applied due to such fundamental differences.

4. Token-based Selective Replay

As the pipeline becomes wider and deeper, current and future microprocessor designs require selective replay to reduce the scheduling miss penalty. Unfortunately, the selective replay schemes discussed in the previous sections have scalability issues and other limitations that prevent many data-speculation techniques from being adopted. This section proposes a token-based selective replay to overcome those difficulties.

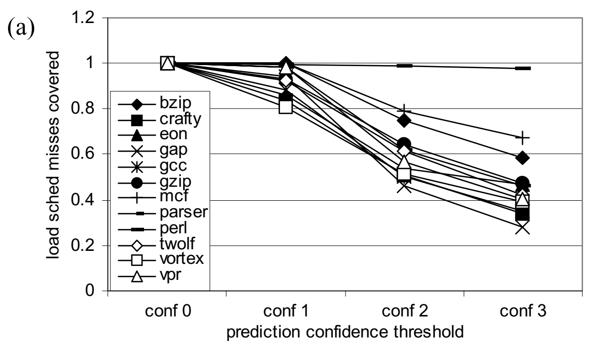
4.1. Reducing the Name Space for Dependence Tracking via Scheduling Miss Prediction

The biggest challenge in implementing selective replay is how to efficiently reduce the name space that identifies dependences among instructions while still providing precise replay. Fundamentally, difficulties with scheduling replay schemes arise from the fact that the dependence information propagates as a part of schedule or execution process. An alternative design choice is to move the propagation loop out of the scheduling logic, and using the rename table for the dependence information propagation as a part of the rename process. At the expense of a larger name space, this approach has several advantages. First, extra wires for propagating dependence information needed in position-based replay scheme are no longer required. Second, propagating dependence information occurs in program order, and correct data dependences do not have to be enforced in the scheduler, implying that the data-speculation techniques that violate data dependences may benefit from this selective replay support. To enable this approach, however, the name space for dependence tracking should be reduced to a reasonable size that the processor can handle in an efficient way.

It is well understood that a small fraction of static loads account for the large portion of dynamic events such as cache misses or memory dependence violations, and that history-based predictors can predict those events. For our experiments, we used a tagged, 4k-entry, direct-mapped table indexed by the PC, with 2-bit saturating counters to dynamically predict load scheduling misses.

Figure 9 shows the coverage of scheduling misses on the left graph, and the fraction of load instructions predicted to be mis-scheduled at a given confidence threshold on the right graph, measured on our 8-wide machine configuration. We note that the ideal predictor for scheduling misses is achieved when the coverage of scheduling misses is 1 and the number of load instructions predicted to be mis-scheduled is no greater than the actual scheduling misses. In general, the coverage of scheduling misses (Figure 9a) decreases but the accuracy of prediction (Figure 9b) improves as we increase the confidence threshold. Among a set of benchmarks we tested, *perl* is measured to be most highly predictable; less than 5% of load instructions are predicted to be mis-scheduled at the highest confidence, and over 95% of all load scheduling misses are covered by our prediction mechanism.

Based on scheduling miss prediction, selective replay can be implemented to consume a smaller name space than required for the worst case if tag names for dependence tracking are allocated only to the instructions that are likely to incur scheduling misses.



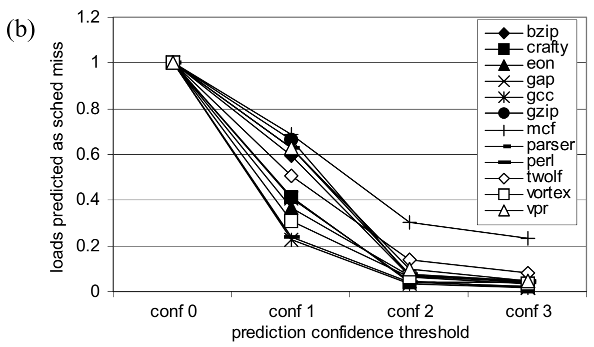


Figure 9. Scheduling miss prediction performance.

4.2. Token-based Selective Replay

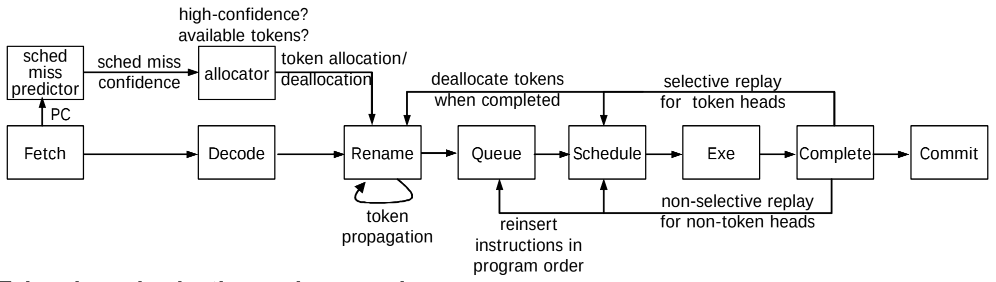
We propose token-based selective replay that enables precise dependence tracking with reduced complexity in the scheduler, and that does not require enforced data dependences. The basic idea is that the token is planted only in instructions that are likely to be mis-scheduled, and they are selectively recovered using tokens. On the other hand, instructions without a token are recovered using re-insert. As long as the scheduling miss predictor provides high coverage, the vast majority of scheduling misses can be selectively recovered and performance is not severely affected by the expensive re-inserts.

Figure 10 illustrates the pipeline overview of the token-based selective replay. The scheduling miss predictor is accessed by PC in parallel with instruction fetch, and generates scheduling miss confidence. The allocator manages a fixed number of unique tokens, and allocates them to load instructions considering confidence level and availability of tokens. Our allocation policy is to eagerly allocate a token to instructions even with low confidence if available, and to reallocate it to a new instruction with higher confidence if all tokens are already used up. For our experiments, 8 and 16 tokens are used for 4 and 8-wide machines, respectively.

Figure 11a illustrates the process of token propagation. This functionality is achieved by adding as many extra bits as the total number of tokens to each rename map entry. When source operands access the rename table for physical register IDs, piggybacked dependence vectors are read from the table. A vector, in which each bit position represents each token, contains the list of parent loads with tokens. The two vectors for source operands are merged and stored back to the target register entry of the rename table. If a token is allocated to a load, this instruction (referred to as *token head*) marks the corresponding bit in the vector before being stored.

When a token head instruction reaches the completion stage, its token ID is broadcast to all instructions in the window, signaling either that the instruction was mis-scheduled or has completed. Since all dependent instructions have the corresponding token in their dependence vectors, this selective replay is precise and the penalty from the scheduling miss is the same as in the position-based selective replay. If the token head has completed successfully, dependent instructions turn off the corresponding bit in their vectors. If all bits are cleared in the dependence vector, the instruction releases the issue queue entry when it is issued or if it has already been issued.

If a load scheduling miss is detected but the instruction is not a token head, it needs to be recovered non-selectively because subsequent instructions cannot track their dependences precisely. In this case, all instructions following the mis-scheduled instruction in program order are flushed out of the issue queue and re-inserted from the reorder buffer, sequentially accessing the map table in program order and checking correct operand status. Although the performance penalty will be quite severe, this provides a safety mechanism that terminates the incorrect speculative wavefront even in the presence of data-speculation techniques, since all in-flight instructions are examined in program order. This *re-insert* replay is similar to what is used in the waiting instruction buffer [17], but our mechanism does not manage a separate instruction buffer to keep track of dependences among instructions. In our simulations, we assume 4 clock cycles of penalty from detecting a scheduling miss to starting re-insert, and instructions are re-inserted at the same rate as the dispatch bandwidth. During the whole re-insertion process, no new instructions are



high-confidence?
available tokens?
sched miss predictor
sched miss confidence
allocator
token allocation/ deallocation
deallocate tokens when completed
selective replay for token heads
PC
Fetch
Decode
Rename
Queue
Schedule
Exe
Complete
Commit
token propagation
reinsert instructions in program order
non-selective replay for non-token heads

Figure 10. Token-based selective replay overview.

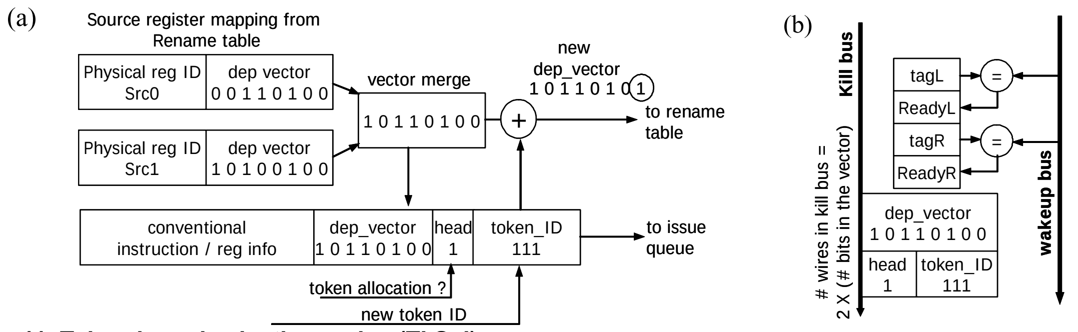

(a)
Source register mapping from Rename table
Physical reg ID Src0
dep vector 0 0 1 1 0 1 0 0
Physical reg ID Src1
dep vector 1 0 1 0 0 1 0 0
vector merge
1 0 1 1 0 1 0 0
+
new dep_vector 1 0 1 1 0 1 0 1
to rename table
conventional instruction / reg info
dep_vector 1 0 1 1 0 1 0 0
head 1
token_ID 111
to issue queue
token allocation ?
new token ID
(b)
Kill bus
tagL
ReadyL
tagR
ReadyR
=
dep_vector 1 0 1 1 0 1 0 0
head 1
token_ID 111
wakeup bus
# wires in kill bus = 2 X (# bits in the vector)

Figure 11. Token-based selective replay (TkSel).

Kill bus (2 bits per token)	State	Description	Operation performed
0 0	idle	don't care	none
0 1	kill	the parent instruction is invalidated	ready bit is cleared
1 0	complete	the parent instruction completes	The corresponding bit in the vector is cleared. The instruction releases the issue entry if all bits in the vector are zero.
1 1	reclaim	This token is not valid anymore, and the token name is reclaimed	The corresponding bit in the vector is cleared, and *token_ID* and *head* fields are cleared if they match the bit.

Table 2: The signal states of the kill bus in the token-based selective replay.

inserted into the window.

When a token head instruction completes, or the token allocator decides to deallocate the token and give it to a new instruction with a higher confidence, the corresponding bit in the dependence vectors in the rename table and instructions should be cleared in order to guarantee that the new token head would not incorrectly invalidate independent instructions from the old mapping. For this purpose, a kill signal for each token consists of two wires that express four states. Table 2 shows these signal states of the kill bus and the operations to perform.

Performance evaluation of token-based selective replay, compared with other replay schemes will be presented in Section 5.4.

5. Simulation Results

5.1. Processor Model

The execution-driven simulator used in this study is derived from the *SimpleScalar* / *Alpha* 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha ISA. Specifically, we extended *sim-outorder* to perform speculative scheduling with various scheduling replay schemes. In this pipeline, instructions are scheduled in the scheduling stage assuming instructions have fixed execution latency, and any latency changes due to cache misses or memory dependences from a store without the store data cause instructions to be rescheduled. We modeled a 13-stage pipeline with 4- and 8-instruction machine width. The pipeline structure is illustrated in Figure 1. The detailed machine configurations are shown in Table 3.

In order to detect memory dependence violation, we apply a policy similar to the one used in [18], which prevents load instructions from being issued if an earlier, unissued store address operation exists. This policy enables a load to check if it is aliased with the any of the previous stores by the time it reaches the completion stage.

5.2. Benchmarks

The SPEC CINT2000 benchmark suite is used for all results presented in this paper. All benchmarks were compiled with the DEC C and Fortran compilers under the OSF/1 V4.0 operating system using -O4 optimization. Table 4 shows the benchmarks, input sets, the number of instructions committed, and IPC on 4 and 8-wide base machines with position-based selective replay. The large reduced input sets from [4] were used for all benchmarks except for *crafty, eon* and *gap*. These three benchmarks were simulated with the reference input sets up to 3 billion instructions since the reduced inputs are not available.

5.3. Base Machine Scheduler Characteristics

Table 5 summarizes the scheduler characteristics on 4 and 8-wide machines with the position-based selective replay, which is the most precise and accurate scheme under the current speculative scheduling constraints, and therefore it is used as the basis for the comparisons presented in this section. The second column (*load scheduling misses / load issues*) in the table presents the load mis-scheduling rates. We note that these numbers are greater than cache miss rates because many scheduling misses come from store-to-load aliasings, and multiple accesses to a newly fetched cache line may be all counted as scheduling misses, whereas only the first access to the line is counted as a cache miss. The last column (*total replays / total issues*) presents unnecessary issues incurred by scheduling misses and their dependent instructions.

5.4. Evaluation of Scheduling Replay Schemes

Table 6 presents the fraction of load scheduling misses that are selectively replayed with tokens in token-based selective replay. We used 8 and 16 tokens for 4 and 8-wide machines, respectively. The structure and performance of the predictor were described in Section 4.1. In this configu-

	4-wide	8-wide
OoO Execution	4-wide fetch/issue/commit, 128 ROB, 64 LSQ, 64 issue queue entries, fetch stops at first taken branch in a cycle	8-wide fetch/issue/commit, 256 ROB, 128 LSQ, 128 issue queue entries, fetch stops at first taken branch in a cycle
Functional Units (latency)	4 INT ALUs (1), 2 FP ALUs (2), 2 INT MULT /DIV (3/20), 2 FP MULT /DIV (4/24), 2 general memory ports	8 INT ALUs (1), 4 FP ALUs (2), 4 INT MULT /DIV (3/20), 4 FP MULT /DIV (4/24), 4 general memory ports
Branch Prediction	Combined bimodal (4k entry) / gshare (4k entry) with a selector (4k), 16 RAS, 1k-entry 4-way BTB, at least 11 cycles taken for misprediction recovery	
Memory System (latency)	32KB 2-way 64B line IL1 (2), 32KB 4-way 64B line DL1 (2), 512KB 4-way 128B line unified L2 (8), main memory (100)	

Table 3: Machine Configurations.

Benchmark	input sets	inst count	Base IPC w/ position-based selective replay (4 / 8-wide)
bzip	lgred.graphic	2.64B	1.6409 / 2.0932
crafty	crafty.in	3B	1.9410 / 2.7949
eon	chari.control.cook	3B	2.1741 / 3.1457
gap	ref.in	3B	2.0737 / 2.8784
gcc	lgred.cp-decl.i	5.12B	1.5148 / 1.9721
gzip	lgred.graphic	1.79B	2.0147 / 2.5117
mcf	lgred.in	0.79B	0.7061 / 0.9225
parser	lgred.in	4.52B	1.2614 / 1.5208
perl	lgred.markerand	2.06B	1.4149 / 1.7067
twolf	lgred.in	0.97B	1.5959 / 1.9205
vortex	lgred.raw	1.15B	2.1217 / 3.1530
vpr	lgred.raw	1.57B	1.6807 / 2.0658

Table 4: Benchmarks.

Benchmark	load scheduling misses / load issues		total replays / total issues	
	4-wide	8-wide	4-wide	8-wide
bzip	3.71 %	6.86 %	2.50 %	4.56 %
crafty	3.16 %	4.06 %	2.50 %	3.19 %
eon	3.05 %	7.77 %	1.44 %	4.00 %
gap	1.67 %	3.86 %	1.10 %	2.03 %
gcc	2.09 %	3.18 %	2.03 %	3.12 %
gzip	4.07 %	5.77 %	3.52 %	4.40 %
mcf	27.59 %	27.60 %	23.02 %	22.45 %
parser	5.91 %	6.81 %	5.08 %	6.05 %
perl	2.31 %	3.71 %	1.10 %	1.51 %
twolf	10.43 %	12.31 %	6.50 %	7.15 %
vortex	4.80 %	6.56 %	2.73 %	4.08 %
vpr	6.86 %	8.88 %	4.68 %	5.58 %

Table 5: Scheduling statistics with position-based selective replay.

Benchmark	4-wide (8 tokens)	8-wide (16 tokens)	Benchmark	4-wide (8 tokens)	8-wide (16 tokens)
bzip	89.7 %	91.9 %	mcf	75.2 %	83.5 %
crafty	88.4 %	89.3 %	parser	85.3 %	88.5 %
eon	88.2 %	91.9 %	perl	99.7 %	99.6 %
gap	91.7 %	95.8 %	twolf	84.9 %	89.5 %
gcc	86.0 %	89.3 %	vortex	90.6 %	93.3 %
gzip	91.8 %	93.6 %	vpr	91.2 %	92.2 %

Table 6: Scheduling misses covered by tokens in token-based selective replay.

ration, 90% of scheduling misses are selectively replayed using tokens on average. *Mcf* shows the lowest coverage since frequent cache misses cause the number of concurrent scheduling misses to exceed the number of tokens. We did not fully explore the design space of the scheduling miss predictor and leave it to future work. However, this coverage can be improved by adopting more advanced designs from prior proposals [8][12][15].

Figure 12 shows issue counts normalized to position-based selective replay. In non-selective replay, the total issue count significantly increases over other selective replay schemes since dependence tracking is not precise and independent instructions are unnecessarily replayed. Moreover, we observe that the issue count is more severely affected as the machine width grows, which may lead to significant impact on performance. In delayed selective replay (*Dsel*), on the other hand, issue count reduction is observed in several benchmarks since this scheme conservatively delays issuing instructions under scheduling miss events. Token-based selective replay shown in the right bars (*TkSel*), almost achieves the issue counts of the base machine except for *mcf* in which over 5% of extra issue bandwidth is consumed due to low prediction coverage. Some benchmarks report issue count reductions in the token-based scheme since the scheduling behavior changes may reduce the total number of scheduling misses.

Figure 13 shows the IPCs of various replay schemes normalized to position-based replay. The first three bars from the left represent the performance of non-selective, delayed-selective (*Dsel*) and token-based selective (*TkSel*) replay schemes discussed in the previous sections. In order to provide insight into scheduling miss predictions, two additional data points are also plotted: the fourth bar shown as *re-insert* is the performance when all scheduling misses are recovered by invalidating and re-inserting instructions in program order. Token-based selective replay relies on this mechanism for the worst-case replay when no token is allocated (as were discussed in Section 4.2); the right-most bar shown as *conservative* represents a case where a load with high scheduling miss prediction confidence (2 and 3) prevents dependent instructions from being speculatively issued until after the actual latency is known. If the prediction is correct (predicted to be a miss and actually a miss), the approach avoids any replay penalty and also benefits from improved resource utilization. If the prediction is incorrect (predicted to be a miss but actually a hit), it will lose performance by unnecessary delaying dependent instructions. This is similar to what is proposed in [8], except that replay is performed using the re-insert replay mechanism when a load is predicted to be a hit but is actually a miss.

With non-selective replay, we observe that the performance degradation is significant and that it does not scale well to a wider machine as the scheduling miss penalty

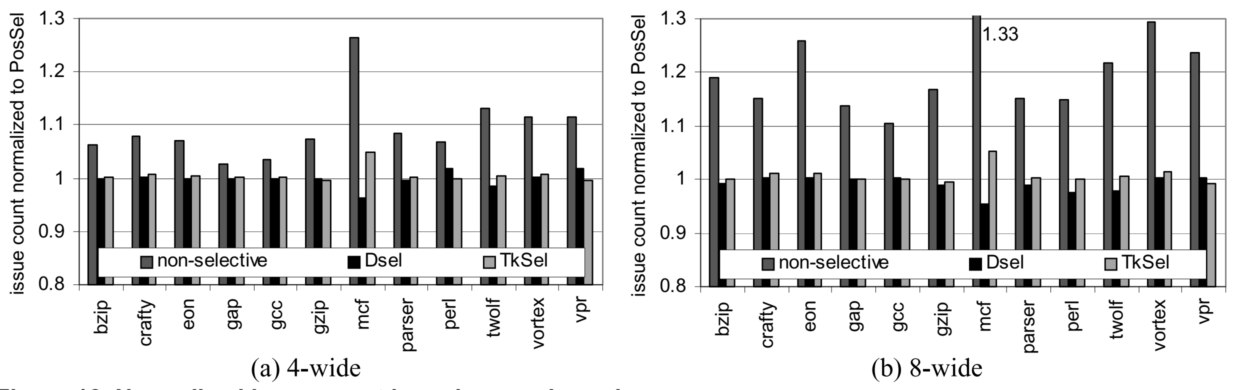


Figure 12. Normalized issue count in various replay schemes.

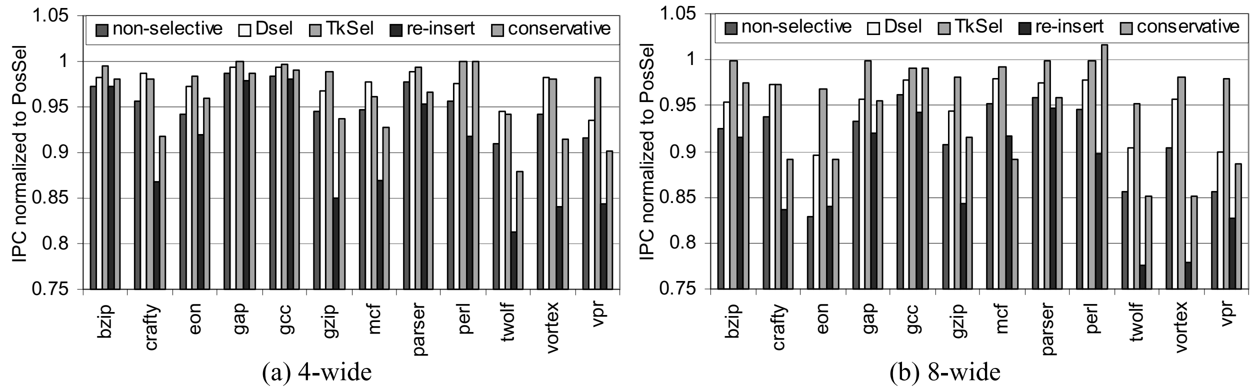


Figure 13. Normalized performance in various replay schemes.

also increases. Delayed selective replay achieves much of ideal performance with relatively simple extensions to the non-selective replay mechanism on the 4-wide machine. However, this scheme also has performance scalability issues because the number of unnecessarily delayed instructions increases as the machine width grows.

In the third bars, we find that token-based selective replay performs similarly to delayed selective replay on the 4-wide machine. However, it performs measurably better on the 8-wide machine because the scheduling miss penalty is not a function of the machine width, and only a small number of scheduling misses trigger re-insert replay. The average slowdown is measured to be 1.7% and 1.6% on 4 and 8-wide machines, respectively.

The prediction-based, conservative scheduling scheme (the right-most bars) shows improved performance over re-insert replay. Ideally, this scheme can perform better than the base position-based replay since it reduces the number of scheduling misses and does not incur penalty when the prediction is correct. We find this case in *perl*, where the predictor (presented in Figure 9) performs almost ideally. However, achieving both high coverage and accuracy for scheduling miss prediction would be difficult in many cases because, even though only a few static instructions incur scheduling misses, they still hit in the cache over a half of the time. This behavior makes the design of a highly accurate predictor challenging, and therefore we believe that an efficient replay mechanism should still be provided to achieve compelling performance.

5.5. Discussion

The results suggest that delayed selective replay would be a good design alternative to ideal position-based selective replay, considering both hardware complexity and performance. Our proposed scheme, token-based selective replay might be overkill for current-generation processor implementations due to complexity over delayed selective replay. However, we find that the performance of both non-selective and delayed selective replay does not scale to wider machines whereas token-based selective replay achieves performance very near to ideal position-based selective replay.

The number of wires in the wakeup and kill buses increases quadratically in the position-based scheme since the size of the matrix and the number of wakeup buses may grow at the same rate. On the other hand, the number of wires for the token-based scheme is a function of the number of tokens, not the machine depth or width. For example, given the 8-wide configuration used in this paper, the total number of extra wires needed for replay is 196 and 32 in position-based and token-based replay schemes, respectively. Of course, our proposed scheme requires other expensive modifications in the pipeline. However, if we assume that the scheduling logic is a major bottleneck limiting processor cycle time and that the size of the wakeup logic is dominated by the number of wires, token-based selective replay would enable faster and more efficient scheduling logic at a marginal performance expense. In addition, token-based selective replay supports data speculation techniques that violate data dependences in the scheduling logic by tracking dependences in order, while other replay schemes cannot recover in cases where data dependences are not carried through the scheduler in correct dependence order.

6. Related Work

The initial proposals on selective re-issue or replay were originally designed for value prediction [10][11], to improve its efficiency by minimizing the misprediction penalty. Many proposals on data speculation techniques in the literature [13][14] also use this replay mechanism for similar reasons. Recovery mechanisms have also been developed for real processor implementations that perform speculative scheduling. The Alpha 21264 [2] uses a scheme that invalidates all instructions in load shadow. The Pentium 4 [3] adopts a more advanced scheme that selectively replays only dependent instructions. Although they rely heavily on efficient replay mechanisms, there are surprisingly few studies on describing how they are implemented in real hardware. Sazeides [9] proposed the generic dependence vector scheme as the selective replay mechanism for value prediction. Morancho et al. [19] evaluated various scheduling replay mechanisms but did not fully describe how dependences are tracked across multiple levels of instructions.

This paper did not explore the whole space of sophisticated designs to predict scheduling misses. The efficiency of the mechanism could be significantly improved by using proposed schemes for predicting memory dependences and cache misses. Moshovos and Sohi [12] proposed a memory dependence prediction technique for improving memory ordering and store value forwarding. Similarly, Austin and Tyson [15] also proposed the speculative memory renaming scheme for improved memory dependence handling. Chrysos and Emer [16] studied memory dependence prediction using store sets. In order to improve load instruction scheduling, Yoaz et al. [8] proposed scheduling miss prediction to schedule loads pessimistically for certain cases.

7. Conclusions

We make three main contributions in this work. First, we describe the problems with incorrect speculative wavefront propagation incurred by the delay between schedule and execution. Second, we explore the design space of scheduling replay schemes, and identify two potential limitations: scalability and data-dependence enforcement. Third, we propose token-based selective replay that moves the dependence information propagation loop out of the scheduler, enabling lower complexity in the scheduling logic and support for data-speculation techniques.

8. Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437, and generous financial support and equipment donations from IBM and Intel. We would like to thank the anonymous reviewers for their many valuable comments.

9. References

[1] D. C. Burger and T. M. Austin, The Simplescalar tool set, version 2.0, *Technical Report CS-TR-97-1342*, University of Wisconsin, Madison, June 1997.

[2] Compaq Computer Corporation, Alpha 21264 microprocessor hardware reference manual, July 1999.

[3] G. Hinton et al., The microarchitecture of the Pentium 4 processor, *Intel Technology Journal* Q1, 2001.

[4] A. KleinOsowski, J. Flynn, N. Meares and D. J. Lilja, Adapting the SPEC2000 benchmarks suite for simulation-based computer architecture research, *Workshop on Workload Characterization in International Conference on Computer Design*, 2000.

[5] J. Stark, M. Brown and Y. Patt, On pipelining dynamic instruction scheduling logic, in *Proc. of 33rd International Symposium on Microarchitecture*, 2000.

[6] I. Kim and M. H. Lipasti, Macro-op scheduling: relaxing scheduling loop constraints, in *Proc. of 36th International Symposium on Microarchitecture, 2003.*

[7] R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal*, Vol. 11, pp. 25-33, 1967.

[8] A. Yoaz, M. Erez, R. Ronen and S. Jourdan, Speculation techniques for improving load related instruction scheduling, in *Proc. of 26th International Symposium on Computer Architecture*, 1999.

[9] Y. T. Sazeides, An analysis of value predictability and its application to a superscalar processor, Ph.D thesis at University of Wisconsin-Madison, 1999

[10] M. H. Lipasti, C. B. Wilkerson and J. P. Shen, Value locality and load value prediction, in *Proc. of Architectural Support for Programming Languages and Operating Systems*, 1996.

[11] M. H. Lipasti and J. P. Shen, Exceeding the dataflow limit via value prediction, in *Proc. of 29th International Symposium on Microarchitecture*, 1996.

[12] A. Moshovos and G. S. Sohi, Streamlining interoperation memory communication via data dependence prediction, in *Proc. of 30th International Symposium on Microarchitecture*, 1997.

[13] A. Moshovos and G. S. Sohi, Speculative memory cloaking and bypassing, in *Proc. of 32nd International Symposium on Microarchitecture*, 1999.

[14] H. Akkary and M. A. Driscoll, A dynamic multithreading processor, in *Proc. of 31st International Symposium on Microarchitecture*, 1998.

[15] T. Austin and G. Tyson, Improving the accuracy and performance of memory communication through renaming, in *Proc. of 30th International Symposium on Microarchitecture*, 1997.

[16] G. Chrysos and J. Emer, Memory dependence prediction using store sets, in *Proc. of 25th International Symposium on Computer Architecture*, 1998.

[17] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan and E. Rotenberg, A large, fast instruction window for tolerating cache misses, in *Proc. of 29th International Symposium on Computer Architecture*, 2002.

[18] Intel Corporation, Pentium Pro family developers manual, Intel corporation, 1996

[19] E. Morancho, J. M. Llaberia and A.Olive, Recovery Mechanism for Latency Misprediction, in *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[20] United States Patent #6,212,626, Computer processor having a checker, Amit A. Merchant and David J. Sager, assigned to Intel Corporation, issued April 3, 2001.

[21] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of superscalar processors*, beta edition, pp. 400-402, McGraw-Hill, 2002.

[22] Y. T. Sazeides, Modeling value speculation, in *Proc. of International Symposium on High-performance Computer Architecture*, 2002.