Practical Selective Replay for Reduced-Tag Schedulers

Dan Ernst and Todd Austin
Advanced Computer Architecture Lab
University of Michigan
Ann Arbor, MI 48109
{ernstd, austin}@eecs.umich.edu

Abstract

The trend towards deeper microprocessor pipelines has made it advantageous or necessary to predict the events that may happen in the stages ahead. A widely-used example of this technique is latency speculation, where the non-deterministic latency of some instructions, such as loads, forces dependents to predict the number of clock cycles these operations will take to complete execution. If there is a misprediction, those dependents that issued speculatively must be restarted or delayed appropriately so that they can execute again with the correct inputs. This process is called a scheduler replay. In the interest of reducing the replay penalty, some recent designs, such as the Pentium 4, have adopted selective replay mechanisms, which reschedule only data-dependent instructions on a latency misspeculation.

The deep pipelining trend has also forced designers to reduce the circuit complexity of individual stages to maintain high clock speeds and to keep power dissipation manageable. Tag elimination [4] is a technique used to reduce the complexity of a processor's issue stage by designing for the average case instruction. In Kim and Lipasti's recent work on Half-Price architectures [9]*, the authors state that it is "impractical" to implement a selective replay mechanism in a machine that uses tag elimination. In this paper, we detail the implementation of a practical selective replay method that is compatible with tag elimination schedulers and discuss the power and performance trade-offs that should be considered when designing a replay system.*

1 Introduction

Recent microprocessor designs have employed increasingly deep pipelines. Breaking the execution core into smaller pieces allows for higher clock speeds and, as a result, higher instruction throughput. Many recent studies [6][8][17] show that more benefit could still be extracted by this technique, indicating that pipelines will likely continue growing longer in the future.

However, the benefits of deep processor pipelines do not come without drawbacks. Placing extra stages in certain segments of the processor may force earlier stages to speculate on the events that may occur later in the pipeline [2]. For example, instruction latency speculation predicts how many cycles a producer instruction will take to complete its execution, which occurs several stages later in the pipeline. This is done so that consumer instructions can be issued to meet their inputs at the optimal time. If the prediction is wrong, however, at least some of the instructions in the stages between issue and execution must be restarted or delayed.

In the interest of keeping the latency misprediction penalty to a minimum, some processor designs, such as the Pentium 4, have included implementations of selective instruction replay. By using this technique, a processor only needs to re-execute instructions which are data-dependent on the misspeculated instruction.

Another pipelining obstacle is that some stages, such as the instruction scheduler, are often too big and slow to fit in a single cycle, and are also particularly resistant to the decomposition process necessary to implement pipelining [14].

Tag elimination [4] was proposed as a solution to an instruction scheduler complexity problem. By tailoring the reservation station structures for the common case of instructions needing to wait for fewer than two inputs, the scheduler tag bus capacitance can be reduced by up to 75%, allowing for a higher clock rate and lower power consumption. In Kim and Lipasti's paper "Half-Price Architecture" [9], the authors introduce a clever selective replay implementation that performs parent-child dependence propagation using the scheduler broadcast busses. In addition, they correctly point out that the combination of tag elimination with broadcast-based selective instruction replay is not practical to implement.

In this paper, we describe selective replay methods that are compatible with tag elimination. Furthermore, we analyze the performance and power consequences of the replay implementation decision.

The remainder of this paper is organized as follows. Section 2 gives background information on tag elimination. In Section 3, we present the different replay mechanisms that are then analyzed in Section 4. Related work is listed in Section 5 and our conclusions are presented in Section 6.

2 Tag Elimination

The techniques presented in [4] draw from the observation that most scheduler tag comparisons are superfluous to the correct operation of the instruction scheduler. Analyses reveal that most instructions placed into the instruction window do not require two source tag compar-

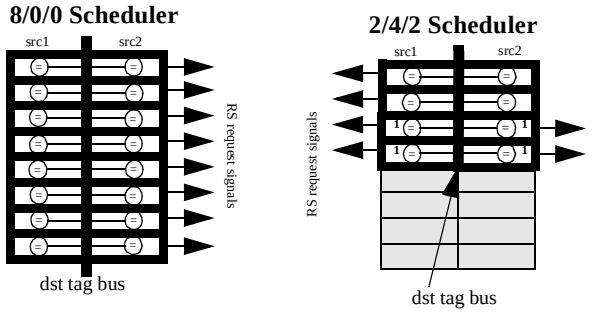


Figure 1. Conventional and Reduced-Tag Reservation Stations. The circles represent tag comparators. The bold tag entries include a comparator. The shaded tag entries are not necessary and thus do not include comparators. Entries with a '1' hold instructions with only one tag to compare against.

ators because one or more operands are ready, or the operation doesn't require two register operands.

Two scheduler tag reduction techniques were proposed that work together to improve the performance of dynamic scheduling, while at the same time reducing power requirements. First, a reduced-tag scheduler design was proposed that assigns instructions to reservation stations with two, one, or zero tag comparators, depending on the number of input operands in flight. An example of a scheduler window using tag elimination is shown in Figure 1.

To reduce tag comparison requirements for instructions with multiple operands in flight, we also introduced a last-tag speculation technique. This approach predicts which input operand of an instruction will arrive last and schedules the execution of that instruction based solely on the arrival of the final operand. Since the earlier arriving tags do not precipitate execution of the instruction, the scheduler can safely eliminate the comparator logic for all but the last arriving operand.

Because scheduling windows that use last-tag prediction don't track the readiness of the operands that are predicted to arrive earlier, a small table must be placed in the following pipeline stage to check that the prediction was correct and all operands actually arrived. This table simply consists of one ready bit for each physical register that is set when values become ready.

3 Implementing Selective Replay

3.1 Parent-Child Broadcast

In the Half-Price Architecture paper [9], Kim and Lipasti present one possible implementation of selective replay. An illustration of this scheme is shown in Figure 2.

Along with its input tags, an instruction's dependence information is kept in the instruction window in the form of one dependence matrix for each input operand. This matrix consists of *W* x *D* bits, where *W* is the machine width, and *D* is the depth of the *load shadow* [2], which is defined as the number of stages between instruction issue and notification of a cache hit or miss. In each matrix position, the presence of a "1" indicates that an instruction is in the corresponding slot in the scheduler pipeline that the current instruction is dependent on, either directly or through some intermediate instructions. Every cycle, the

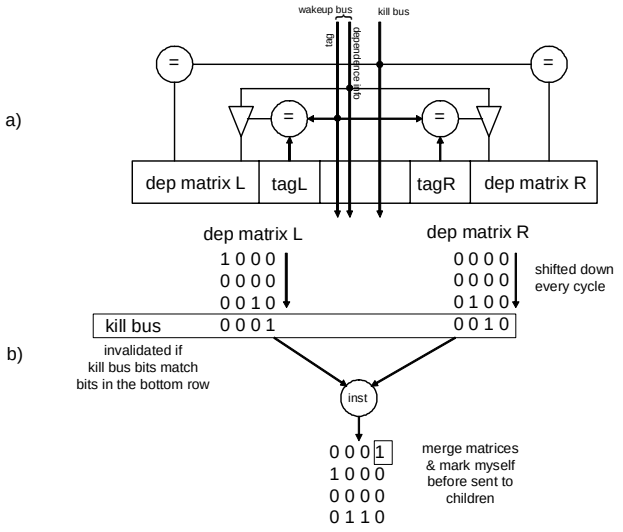


Figure 2. Half-Price Selective Replay Mechanism. (Figure from [9])

matrix shifts down one row, to keep the information consistent with the movement of instructions through the pipeline ahead.

When a load latency misprediction occurs, the execute stage sets the appropriate bit in a *W*-bit wide array called the *kill bus*. These bits are broadcast to every instruction in the scheduler. As is illustrated in part b) of Figure 2, if the kill bus has a 1 in the same column as a 1 in the bottom row of a tag's dependence matrix, that operand's ready bit is reset to zero, as it is dependent on the instruction with the mispredicted latency. In other words, the matching bits indicate that the operand is dependent, either directly or indirectly, on the mis-scheduled instruction.

When an instruction leaves the scheduler window, it merges the dependence matrices that its input operands have received from their parent instructions and marks its own location. It then broadcasts this matrix, along with its destination tag, to the rest of the instructions in the window. (It must also write these bits into a table in the register rename stage for the benefit of dependent instructions that may have not entered the window yet.) When instructions in the window match the input operand on the tag bus, they also latch the dependence matrix of the broadcasting parent instruction. This process propagates dependence information from parent to child during the wakeup phase, giving them knowledge of ancestor instructions further up the dependence tree.

3.1.1 Tag Elimination Compatibility

As is discussed in [9], this selective replay scheme is not compatible with reduced-tag schedulers. Because reduced-tag scheduling makes decisions based on operand availability, problems arise when this availability information is allowed to change after instructions enter the window. Broadcast-based replay relies on every operand in the window tracking its dependencies. Because reduced-tag schedulers gain their complexity benefit by removing some operands from the tag bus, this is not possible.

In schedulers that use non-speculative tag elimination (*i.e.* they do not use last-tag speculation), instructions

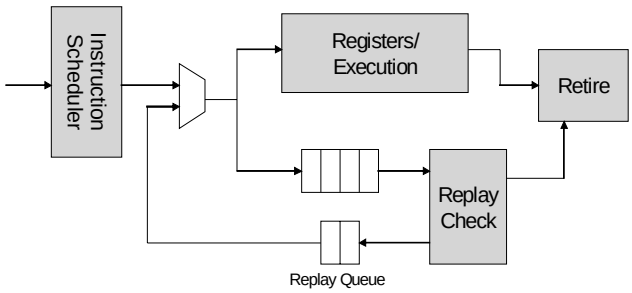


Figure 3. Intel Selective Replay Mechanism.

entering the window would get the proper dependence matrices from the table in the rename stage and they would still be able to monitor the kill bus. However, if the ready bit of an operand that has no comparator needed to be reset, there would be no way for that operand to return to snooping the tag bus.

Furthermore, removing the early arriving operands from the tag bus in last-tag speculation windows makes it impossible for those operands to receive their propagated dependence information from a broadcasting parent instruction.

3.2 Replay using Timed Queues

There are, however, other ways to implement selective replay. Some of these mechanisms differ from the parent/child broadcast model in that, instead of re-executing dependent instructions, they insert a delay into an instruction's execution latency, often through use of a queue or other type of separate instruction storage. The specific mechanism we outline here is derived from a technique proposed in U.S. Patent 6,212,626 [12], held by Intel for inventors Merchant and Sager of the Pentium 4 architecture team [7].[1] A block diagram of this design is shown in Figure 3.

As instructions approach execution, they are also processed through a replay check, which consists primarily of a table of register ready bits. Just before entering the execute stage, instructions look up the status of their input operands in the checker. If the table indicates that the operands are ready, the instruction is allowed to enter execution and retire normally.

If, however, the check table indicates that an operand is unavailable, the instruction sets its output operand as not ready in the table and returns to execution via a replay queue and mux. The replay mechanism informs the scheduler of the presence of an approaching replayed instruction, so that nothing is scheduled into that slot in the same cycle. It is important to note that, in order to maintain forward progress, replaying instructions must always have priority over any work that would be coming out of the instruction scheduler.

When the replayed instruction reaches the input mux, it is sent back into the pipeline as if it had just been issued by the scheduler. On reaching execute, it checks its operands again, just as it did before, to determine whether it needs to replay again (An instruction may have to replay several times to tolerate an L2 cache miss, for example).

In this scheme, the propagation of dependence information is accomplished by the cascading ready bit manipulations in the check table. If there is a latency misprediction, the offending instruction's output will not be set as ready, which will trigger a replay for its children, which in turn will cause a replay for its children's dependents.

It is not specified in the patent exactly how many issue slots coming from the scheduler are stopped when an instruction replays. In our evaluation, we only prohibit the scheduler from issuing into the specific slot that the replaying instruction will be using. This allows the scheduler to issue instructions in the other issue slots.

3.2.1 Tag Elimination Compatibility

A key feature of the Intel replay mechanism is that it maintains the relative timings of instructions throughout the replay sequence. Once a mis-speculated instruction completes, its dependents are replayed just as they were originally scheduled out of the window, only the entire stream has been delayed to accommodate the unexpected extra latency. Consequently, there is no need to "re-schedule" instructions individually, as the previously selected schedule is still valid.

Because replayed instructions are not returned to the scheduler window, there is no extraneous dependence information kept in the window itself. Therefore, reduced-tag schedulers are fully compatible with the Intel-style selective replay.

In schedulers that use last-tag speculation, a last-tag misprediction still results in a one-cycle flush. This recovery is necessary to stop the wakeup of instructions dependent on the last tag misprediction.

4 Replay Evaluation

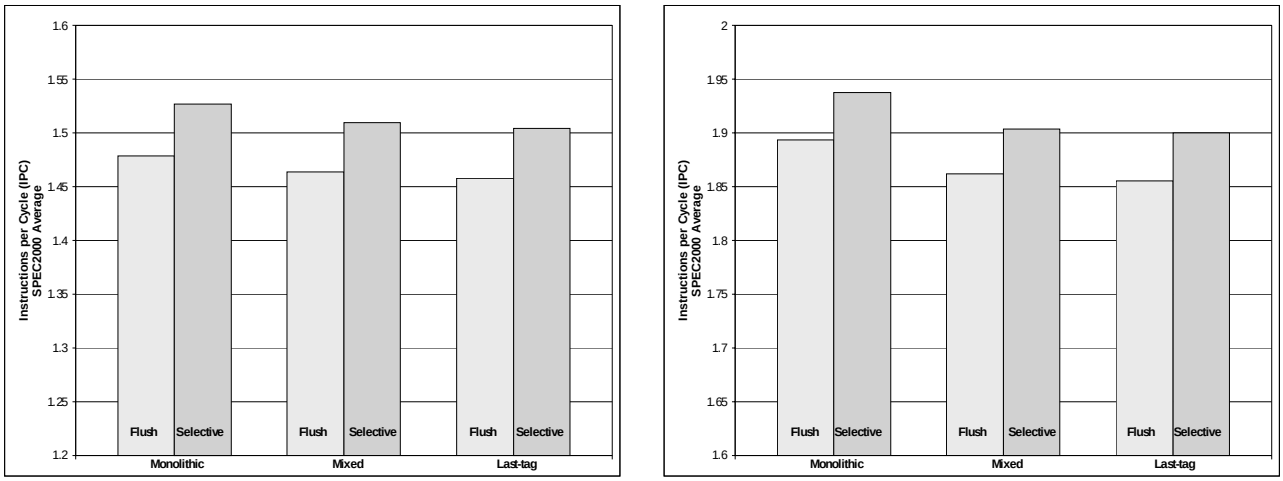
4.1 Simulation Methodology

The architectural simulators used in this study are derived from the SimpleScalar/Alpha version 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction.

To perform our evaluation, we collected results from all 25 of the SPEC2000 benchmarks [18]. All SPEC programs were compiled for a Compaq Alpha AXP-21264 processor using the Compaq C and Fortran compilers under the OSF/1 V4.0 operating system using full compiler optimization (-O4). The simulations were run for 100 million instructions using the SPEC reference inputs. We used the SimPoint toolset's Early SimPoints [16] to pinpoint program locations to simulate for peak accuracy. Simulation parameters are shown in Table 1.

The simulators were modified to separate reservation stations from the re-order buffer. They were also given support for reduced-tag windows and last-tag prediction. Finally, we were able to simulate either 21264-style flush replay [3] or selective replay using one of the two methods outlined in Section 3.

1. Although we may refer to this as the "Intel" replay mechanism throughout this work, we are making no claim as to whether or not this technique is used in any of their microprocessors, commercial or otherwise. We are only presenting the idea proposed in the publicly available patent documentation.



Issue width 4 **Issue width 8**

Figure 4. Effect of Selective Replay on Reduced-Tag Schedulers.

Table 1. Simulation Parameters

Parameter	Values
Execution	256-Entry ROB 4- or 8- wide issue 128-Entry Instruction Scheduler Window "Mixed" windows have 1/2/1 ratio "Last-tag" windows have 0/3/1 ratio replay with 4-cycle *load shadow*
Function Units	8 Integer ALU/MULT/DIV, 4 memory ports, 8 FP ALU/MULT/DIV
Branch Prediction	8k entry GSHARE with 8 bits of global history 2K entry BTB, 8-entry RAS
Last Tag Prediction	4k entry GSHARE (only for "Last-tag" configurations) 1-cycle flush misprediction penalty
Memory System	32KB 4-way associative L1 Instruction and Data Caches with 1-cycle latency, 256KB 4-way associative unified L2 with 16-cycle latency, 100 cycle main memory latency across a 16-byte bus

4.2 Tag Elimination and Replay

The SPEC benchmarks were simulated with three different schedulers on both 4- and 8-wide issue configurations, with the results shown in Figure 4. The baseline scheduler ("Monolithic") and the reduced-tag schedulers ("Mixed" and "Last-tag") all gain 2-3% performance improvement due to the decreased replay penalty. *Galgel* benefitted the most with a 26% improvement due to a large number of memory references and enough parallelism to suffer from pipeline flushes. No benchmarks saw a performance degradation due to selective replay. The performance improvement of 2-3% would close much of the gap demonstrated in the experiments of Kim and Lipasti [9].

4.3 Instruction Window Pressure

As is discussed briefly in Borch *et al.*'s work [2], the parent-child broadcast replay model requires that instructions must remain in the scheduler window for several cycles after they are issued, in order to monitor the kill bus for a replay indication. When all of an instruction's ancestors are safely into execution, only then will it finally release its reservation station.

As a result of keeping instructions in the window beyond their issue time, this mechanism can suffer from a reduction in effective scheduler window size. For example, an 8-wide machine with a 4-cycle *load shadow* could be holding as many as 32 instructions in the scheduler window that have already issued, reducing the number of spots that are available for newer instructions. For the typical case, the number of extra instructions held in the window will not be that large because the processor will not usually be filling all of its issue slots.

Using the Intel replay technique, instructions never re-execute out of the scheduler window, thus removing the need to stockpile instructions after they've issued. This reduces the instruction pressure in the window, allowing more work to flow into the empty slots.

On the other hand, the Intel approach can limit execution bandwidth when too many instructions are in replay, thus preventing new instructions from entering execution. However, if there is a large number of replaying instructions, either they are all waiting for one long-latency load, or there are multiple outstanding latency mispeculations. In either case, it is not likely that much more parallelism could be found anyway.

Schedulers with 32, 64, and 128 entries were simulated using both replay techniques, with the results shown in Figure 5. As intuition would suggest, the most benefit was seen in configurations with smaller windows and wider issue, with the 32-entry 8-wide scheduler receiving a 5% performance improvement from the reduced instruction window pressure.

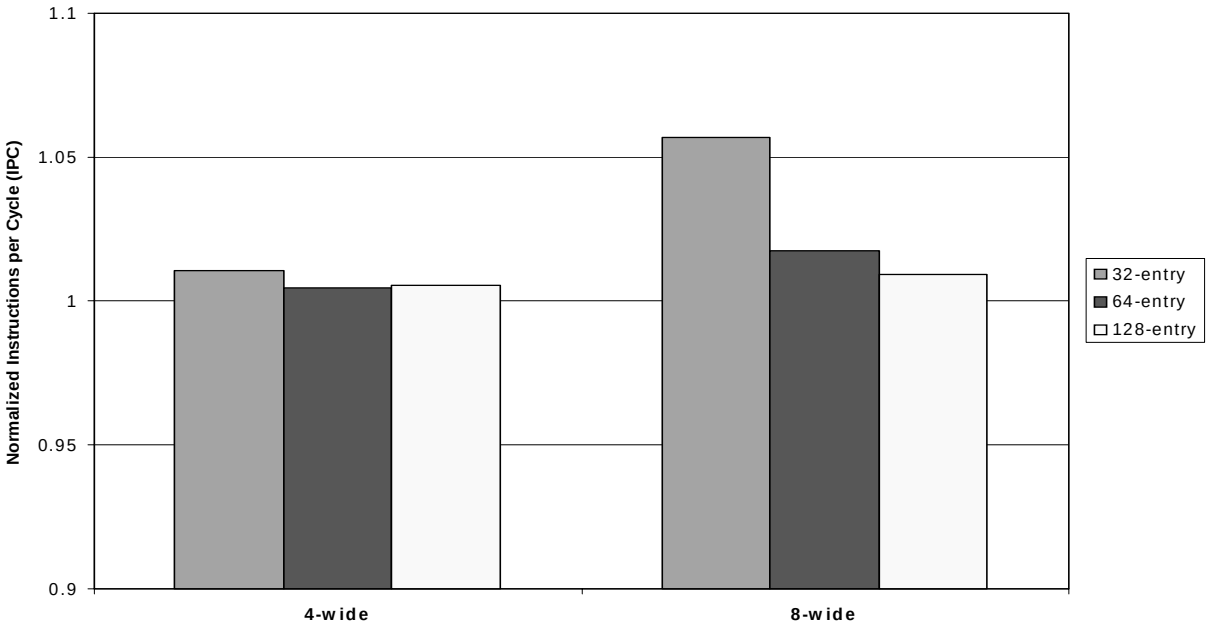


Figure 5. Effect of Reduced Scheduler Pressure. Results presented show the relative performance of a scheduler using the Intel replay mechanism with respect to a scheduler of the same size with a broadcast-based mechanism.

4.4 Power Consumption

In the parent-child broadcast mechanism, the dependence matrix of an issuing operation is sent to all other instructions in the scheduler window. It has been shown in previous studies [4] that these wire-intensive broadcasts can be very costly from a power standpoint.

The load on the broadcast bus as seen by each dependence matrix bit can be estimated as roughly equivalent to that seen by each destination tag bit. While each bit of the matrix bus could be separated from the matrix latches by a low-capacitance pass gate, the bus line must still be able to drive the input of the latch if this gate is open. While having all of the pass gates open is an extreme case (all instructions depend on the broadcast through both operands), it is necessary to take it into account as a peak case. The kill bus bits will also have the same load as the tag bits, since they are also being driven to comparators for each operand.

In a standard instruction scheduler without this replay mechanism, the number of bits broadcast each cycle is

W x (dest tag bits),

where *W* is the scheduler issue width and the number of destination tag bits is $log_2(\#\ of\ physical\ regs)$. If the parent-child broadcast mechanism is incorporated, the number of bits broadcast each cycle is

W x (dest tag bits) + (W x (W x D)) + W,

where *D* is the number of cycles in the load shadow. The first portion of the equation represents the destination tag broadcasts, and it is the same as for the standard window. The second and third terms of the equation represent the dependence matrix bits and the kill bus bits, respectively.

This drastic increase in broadcasts may not directly alter the cycle time (although the layout expansion could have some effect). However, the power consumption will likely be noticeably larger. For example, an 8-wide window with a load shadow of 4 and 256 registers will need to broadcast 328 bits across the scheduler instead of just 64.

The Intel replay technique requires none of these extra broadcasts. The mechanism does include some extra logic, but the power consumed by the check table should be less than the amount that would be dissipated across wire-intensive broadcast lines. This comparison is similar in scope to the comparison of the power usage of a last-tag predictor table with the power used by the scheduler window given in [4].

5 Related Work

Several researchers have recently made the observation that benefit can be gained from removing long latency instructions from the scheduler window as soon as possible. LeBeck's WIB scheduler identifies instructions dependent on long latency operations (data cache misses), and directs these operations to a secondary scheduler [10]. When the long latency operation nears completion, the dependent operations are dumped *en masse* into a small CAM-based dynamic scheduling window. Morancho used a similar approach to move dependent operations following long latency instructions out of the instruction window [13]. Unlike the WIB, they record relative instruction latencies to simplify the re-execution of operations once a valid schedule has been built. The Intel mechanism utilizes a similar approach. As instructions replay, dependencies between dependent operations are maintained by their spacing in the scheduler queues. A schedule is picked and fully committed to for the lifetime of the instruction. Our recent work on the Cyclone scheduler [5] takes this method a step further and replaces the scheduler window

with a dataflow pre-scheduler and timed queues. A queue-based replay mechanism is relied upon to accommodate any incorrectly scheduled instructions.

A number of previous efforts have utilized the register forwarding infrastructure to initiate selective instruction re-execution. The sentinel scheduling technique [11] used "poison bits" contained in the register file that were set when load instructions faulted or did not complete. A branch back to the start of the faulting code would then selectively re-execute the faulting code sequence. As instructions read their registers, only those instructions with poison operands needed to re-execute. The approach is quite similar to the Intel replay queue approach, except instead of redirecting program control, instructions themselves are redirected back into the replay queue. Poison bits were employed in a similar manner by Rogers [15].

6 Conclusions

In the interest of minimizing the performance penalties of deep pipelining, modern processors include selective replay mechanisms to reduce the number of instructions lost due to latency mispredictions. Because these designs may also wish to use complexity-reduction techniques such as tag elimination to improve the performance of dynamic scheduling, it is important for the selective replay implementation to be as unintrusive as possible to the instruction window.

The Intel-style selective replay allows for optimizations such as tag elimination by having its mechanism almost completely external to the instruction scheduler structure. In addition, selective replay mechanisms that are queue and table based have the benefits of both less instruction pressure on the issue window and favorable power characteristics.

Acknowledgements

This work was supported the National Science Foundation CADRE program, grant no. EIA-9975286, and by a National Science Foundation CAREER award, grant no. CCR-0093044. Equipment support was provided by Intel Corporation.

References

[1] Todd Austin, Eric Larson, Dan Ernst. SimpleScalar: an Infrastructure for Computer System Modeling, *IEEE Computer*, Volume 35, Issue 2, Feb. 2002.

[2] E. Borch, E. Tune, S. Manne and J. Emer. Loose Loops Sink Chips, In *Proceedings of the 8th International Symposium on High Performance Computer Architecture*, January 2002.

[3] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, June 2001.

[4] Dan Ernst and Todd Austin. Efficient Dynamic Scheduling through Tag Elimination, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[5] Dan Ernst, Andrew Hamel, and Todd Austin. Cyclone: A Broadcast-free Dynamic Instruction Scheduler with Selective Replay, In *Proceedings of the 30th International Symposium on Computer Architecture* (ISCA-30), June 2003.

[6] A. Hartstein and T. Puzak. The Optimum Pipeline Depth for a Microprocessor, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[7] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, Patrice Roussel. The Microarchitecture of the Pentium 4 Processor, In *Intel Technology Journal*, 1st quarter 2001.

[8] M. Hrishikesh, N. Jouppi, K. Farkas, D. Burger, S. Keckler, and P. Shivakumar. The Optimal Logic Depth per Pipeline Stage is 6 to 8 FO4 Inverter Delays, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[9] Ilhyun Kim and Mikko Lipasti. Half-Price Architecture, In *Proceedings of the 30th International Symposium on Computer Architecture* (ISCA-30), June 2003.

[10] Alvin R. Lebeck, Jinson Koppanalil, Tong Li, Jaidev Patwardhan, and Eric Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses, In *Proceedings of the International Symposium on Computer Architecture* (ISCA-29), May 2002.

[11] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors, *ACM Transactions on Computer Systems*, 11(4):376--408, 1993.

[12] 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.

[13] E. Morancho, J.M. Llaberia, A. Olive. Recovery Mechanism for Latency Misprediction, In *Proceedings of the 2001 International Symposium on Parallel Architectures and Compilation Techniques* (PACT-2001), September 2001.

[14] S. Palacharla, N. P. Jouppi, and J. Smith. Complexity-effective Superscalar Processors, In *Proceedings of the 24th Annual International Symposium on Computer Architecture* (ISCA-24), June 1997.

[15] Anne Rogers and Kai Li, Software Support for Speculative Loads, In *Proceedings of the Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '92), 1992.

[16] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2002), October 2002.

[17] Eric Sprangle and Doug Carmean. Increasing Processor Performance by Implementing Deeper Pipelines, In *Proceedings of the 29th International Symposium on Computer Architecture* (ISCA-29), May 2002.

[18] Standard Performance Evaluation Corporation, http://www.specbench.org.