

Complexity-Effective Issue Queue Design Under Load-Hit Speculation*

Tali Moreshet and R. Iris Bahar

Brown University, Division of Engineering, Providence, RI 02912
{tali,iris}@lems.brown.edu

Abstract

Current trends in microprocessor designs indicate increasing pipeline depth in order to keep up with higher clock frequencies and increased architectural complexity. Speculatively issued instructions may be particularly sensitive to increase in pipeline depth, assuming that issued instructions are kept in the issue queue. In this paper, we evaluate the effectiveness of load hit speculation as pipeline depth increases. Effectiveness is measured in terms of performance improvement, issue queue size requirements and re-issue policy. Our results indicate that load hit speculation increases the percentage of issue queue instructions that are waiting to be re-issued, or replayed. This trend increases even more as pipelines become deeper. We propose an alternative, complexity-effective design for the issue queue, that takes into consideration the different utilization that load hit speculation demands from the issue queue.

1 Introduction

Current modern superscalar processors rely on execution of instructions out of program order to enable more instruction level parallelism, and higher performance. In order to be able to execute more instructions per cycle, it is necessary to minimize false dependencies among instructions, hide instruction latencies, and predict latencies of instructions. In particular, load instructions pose several limitations. One problem is memory dependency, which occurs when a load instruction reads from a memory location to which a previous store instruction wrote. The memory address computation result is ready after schedule time, and therefore prevents the early issue of load instructions. This problem was addressed in [18], [5], and [6].

Another problem associated with load instructions is the scheduling of instructions dependent on load instructions. In general, there is a non-zero delay time between the point an instruction is issued to the time it can begin execution. This delay is due to register file access and moving of data across buses. Early issue of instructions dependent

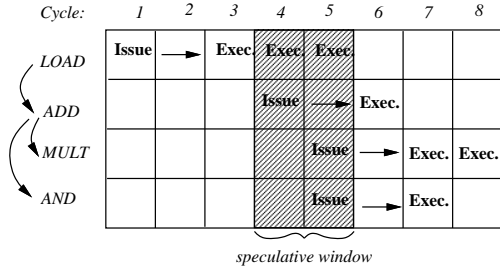
on a load is problematic since load instructions have a non-deterministic latency due to their unknown hit/miss status. The load resolution loop is the delay between the issue of a load instruction until its hit/miss information (also referred to as the hit signal) is passed back to the load's dependent instructions. This loop delay increases as the delay between instruction issue and execute increases.

There are a few options to deal with the latency of the load resolution loop. The conservative approach requires that all instructions dependent on a load value delay their issue time until after the load instruction accesses the cache, to determine if it hit in the first level cache. This approach causes a loss in performance, since most of the load instructions actually hit in the first level cache. The opposite approach allows all instructions following a load to issue early, assuming that the load hits in the cache, and therefore has minimal latency. The second approach pays a penalty in performance for the re-execution of all of the instructions dependent on a load that actually missed in the cache. One way of enabling this re-execution is to keep all instructions in the issue queue until the load hit status is known, in case some of those instructions may be required to re-issue after a load miss. Alternatively, we may use some other means of reinserting the instructions back into the issue queue, by re-fetching all instructions after a load miss. This would eliminate the need to keep post-issue instructions in the issue queue, but according to [3] the performance penalty would be too great to consider this approach.

Figure 1 shows how instructions would flow through a pipeline when loads are speculated to hit in the first level cache. In this example, the *ADD* is dependent on the *LOAD* while the *MULT* and *AND* instructions are dependent on the *ADD* result. For this example, we assume a 2 cycle latency between the time an instruction issues to the time it begins execution. Once execution begins, loads take 3 cycles before hit status is known. As shown in the figure, the *ADD*, *MULT* and *AND* instructions are all issued speculatively during cycles 4–5 before it is known whether the load actually hit in the cache. The gray area in the figure indicates the *speculative window* from which any instruction issued during this time may need to be re-issued, or *replayed*

*This work was supported in part by an NSF-CAREER grant number MIP-9734247.

Figure 1. Instruction flow in a pipeline using load hit speculation. Gray area indicates speculative window.



if the load is discovered to have missed in the cache.

The Alpha 21264 [7], as well as the Pentium4 [12] processor use load hit speculation. The Alpha 21264 allows instructions dependent on a load instruction to be issued assuming the load instruction hit in the first level cache, and therefore has minimal latency (that of the first level cache access). If the load hits, then its dependent instructions benefit from the possibility of issuing early. If the load misses, the wrongly issued instructions need to be re-issued. The Alpha has separate integer and floating point pipelines, each with a different sized speculative window. If an integer load instruction misses, then once the miss is discovered, all the instructions issued after the load, regardless of whether they are dependent on it, are replayed. For a floating point load instruction, only the instructions dependent on the load will be replayed in case of a load miss. Replay of instructions is done by aborting instructions as soon as it is discovered that a load missed in the cache, and after instructions are aborted, they are allowed to request service again.

Current trends in microprocessor designs show increasing pipeline depth in order to keep up with higher clock frequencies and increased architectural complexity. High clock frequencies allow fewer levels of logic to fit within a single clock cycle, even with improved device speed. Also, increasing complexity of logic and data structures may require more pipeline stages. With load hit speculation, deeper pipelines will affect the size of the speculative window since this implies a longer load resolution loop. In addition, this larger speculative window may in turn increase the demands on the issue queue. In particular, if post-issue instructions are retained in the issue queue until the load hit status is known, a larger part of the issue queue will be filled up by these instructions. Unless there is a miss in the cache, these post-issue instructions will not be candidates for selection. These instructions add complexity to the issue selection logic, which is directly related to the size of the queue.

In this paper, we evaluate the effectiveness of load hit speculation as pipeline depth increases. We also consider a

few variations of load hit speculation including re-execution policies and load hit/miss prediction. We show that with load hit speculation, more instructions are post-issue per clock cycle, limiting the effective utilization of the issue queue structure. Furthermore, as was pointed out in [1], today's designs may scale poorly with technology, thus requiring designers to select among deeper pipelines, smaller structures, and/or slower clocks to maximize performance. To account for these trends, our study also considers the relationship of the issue queue size and its latency in order to support load hit speculation and deeper pipelines. We measure the added complexity of load hit speculation in terms of size and timing requirements for the issue queue. Finally, we propose a complexity-effective issue queue structure that separates the post-issue instructions from the rest of the pending pre-issued instructions in the queue, thus allowing the issue queue size to grow without increasing its critical path latency.

The rest of the paper is organized as follows. Section 2 presents the implementation of load hit speculation in the simulation model and simulation techniques. Section 3 provides the results of our simulations for load hit speculation in terms of performance, and discusses the effects on the issue queue. Section 4 discusses the impact of the issue queue design on performance. Section 5 discusses the reasoning behind the design modifications and describes the modifications made to the issue queue. Section 6 lists related work previously done in this area. Section 7 concludes the paper.

2 Load Hit Speculation Model

The simulator used in this study is a modified version of the SIMPLESCALAR [4] tool suite. The configuration of the processor models a future generation out-of-order micro-architecture: The processor has an 8 instruction wide pipeline, and a relatively large number of execution units to allow the full use of the pipeline width. In addition, we implement a separate reorder buffer (ROB) and issue queue (ISQ). The caches of the processor are relatively small to allow for variable cache miss rates to the data cache, in order to demonstrate the effect of load hit speculation on various types of applications. Our simulator models a single unified queue for integer and floating point instructions, and we assume that issued instructions are kept in the issue queue until it is known that they will not need to be reissued. Table 1 shows the complete configuration of the processor.

The processor uses the conservative approach of memory dependency prediction; loads can only execute when all prior stores addresses are known. Also, all stores are issued in program order with respect to prior stores. Although using any kind of memory dependence prediction as suggested in [6] would probably have improved the performance, we chose to limit the prediction done in this study

Table 1. Baseline processor configuration.

Parameter	Configuration
Inst. Window	64-entry LSQ, 256-entry ROB 64-entry ISQ
Machine Width	8-wide fetch, issue, commit
Fetch Queue	16-entry
Number of FUs Latency in ()	8 Int add (1), 2 Int mult/div (3/20) 4 Load/Store (3), 8 FP add (2) 2 FP mult/div/sqrt (4/12/24)
L1 Icache	16KB 2-way; 32B line; 1 cycle
L1 Dcache	8KB direct; 32B line; 3 cycle
L2 Cache	128KB 4-way; 64B line; 12 cycle
Memory	16 bit-wide; 24 cycles on hit, 50 cycles on page miss
Branch Pred.	4k 2lev + 4k bimodal + 4k meta 6 cycle mispred. penalty
BTB	1K entry 4-way set assoc.
RAS	32 entry queue
ITLB	64 entry fully assoc.
DTLB	64 entry fully assoc.
Backend	variable pipeline depth

to load latency.

Modifications to SIMPLESCALAR also include a variable, user-defined delay between the issue and added execute stage, in order to increase the pipeline depth specifically between the issue and writeback stages. Also, a variable wire latency was added for the feedback of hit/miss information from the cache to the consuming, post-issue instructions. In addition, dependency information of issued instructions is broadcast to consuming instructions that are residing in the issue queue after the producing instructions are issued, rather than when they reach the writeback stage. Consuming instructions may be issued such that the producers' results will be ready by the time execution begins. Load instructions are speculated to be ready after the time it takes to access the level 1 data cache. If a load missed in the cache, all the instructions that were issued speculatively need to be removed from the pipeline, and replayed once the load reaches the writeback stage (i.e., once the load data is available). A few methods of replaying instructions are available:

Off: Wait until the writeback stage for resolving output dependencies of all loads (i.e., assume that all loads miss in the cache). For non-load instructions, their issue queue entries are released after they issue and resolve output dependencies. For load instructions, their issue queue entries are released when they reach the writeback stage.

Perfect: The latency of a load access is known in advance. Only loads that miss in the level-1 cache will wait until the writeback stage for resolving dependencies, at

which point they can be released from the issue queue. All other instructions can be released from the issue queue immediately after they issue.

Dependent: Speculate that all loads hit in the first level cache. Replay only instructions that were issued after a mispredicted load and are dependent, directly or indirectly, on the load. Issue queue entries are released only when instructions reach writeback, for all instructions.

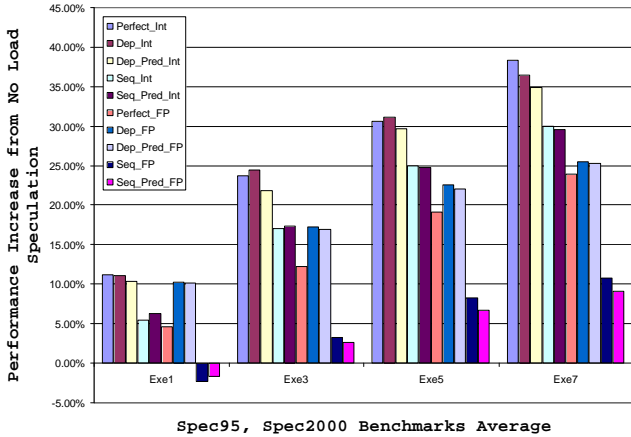
Sequential: Speculate that all loads hit in the first level cache. Replay all instructions that were issued after a mispredicted load, dependent on the load or not. Issue queue entries are released only when instructions reach writeback, for all instructions.

Additionally, we implemented a load hit/miss predictor similar to the Alpha 21264 [8]. The predictor we used is a global 4 bit counter that is decremented by 2 for each load miss and incremented by 1 for each load hit. If the most significant bit of the predictor is 1, then the next load is predicted to hit in the cache. This method minimizes latencies in applications that often hit in the cache, and avoids the costs of over-speculation for applications that often miss. This predictor was chosen since it is simple to implement, and is space and energy efficient. It was used both with the dependent method of replaying instructions and with the sequential method.

Simulations are executed on a subset of the SPEC95 and SPEC2000 integer and floating point benchmarks [9], [11]. All benchmarks are fast-forwarded for 50 million instructions to avoid startup effects, then executed for 100 million committed instructions, or until they complete, whichever comes first. All inputs come from a reference set.

In the sequential replay mode, some restrictions were made on the issue of load instructions. In this mode, in case of a mispredicted load, all instructions issued after the load are replayed. Among the replayed instructions may be other mispredicted load instructions, which will then need to be replayed, as well as all the instructions issued following those loads. As a result, some instructions may be replayed more than once, and this may cause a deadlock in issue of instructions, caused by false dependencies. In order to avoid deadlocks, issue of loads following a mispredicted load was partially blocked. That is, some load instructions are blocked from issuing while there is a load pending in the pipeline.

Figure 2. Performance change of using load hit speculation for different speculation schemes and varying pipeline depths.



3 Effect of Load Hit Speculation with Deeper Pipelines

3.1 Performance Advantage

As a baseline for comparisons, we started by running simulations with no load hit speculation, and a d-cache latency of 3 cycles. That is, we used a conservative approach that assumed all loads miss in the cache. This required waiting until they reach the writeback stage to issue dependent instructions. We also ran simulations with perfect load hit speculation, under a range of pipeline depths, to see whether load hit speculation is at all beneficial. In this case it is known in advance for each load instruction if it will miss in the cache. Dependent instructions are issued at the earliest point possible assuming advanced knowledge of the load latency. All simulations were run with latencies of 1, 3, 5 and 7 cycles between the issue and execution of instructions.

On average, the behavior of the integer benchmarks differed from that of the floating point benchmarks. Figure 2 shows the increase in IPC of the different load hit speculation types, in comparison to no load hit speculation. As the pipeline depth increased from **Exe1** through **Exe7**, the integer benchmarks showed an improvement in performance between 11–38% with perfect load hit speculation (see bars marked **Perfect_Int**). The floating point benchmarks showed a less dramatic improvement between 4–24% as the pipeline depth increased (**Perfect_FP**). The reason floating point benchmarks do not show as great an improvement is that these benchmarks tend to have more parallel streams of dependent instructions. A load miss only affects that particular load’s stream of instructions. Thus, even if one stream stalls until load hit status is known, enough parallelism may

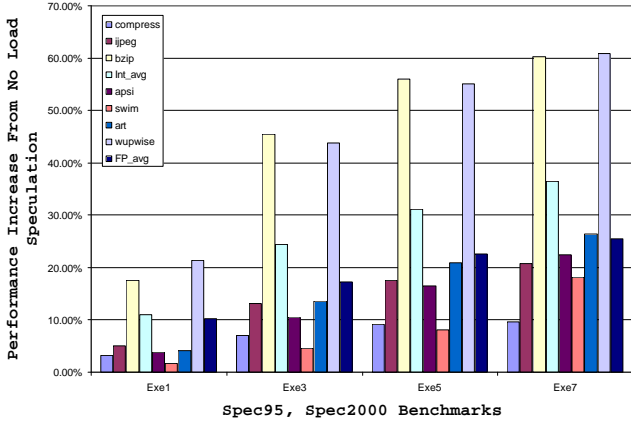
exist such that the extra latency is effectively hidden, and the effect on performance is less dramatic.

Load hit speculation with dependent replay of instructions had performance improvement very close to that of the perfect prediction for both the integer benchmarks (**Dep_Int**) and the floating point benchmarks (**Dep_FP**). In some cases, dependent replay of instructions slightly outperformed the perfect prediction. We suspect this is due to the fact that the issue ordering of instructions changes with dependent replay of instructions, which may affect performance. We are currently investigation this behavior further. The load hit/miss predictor did not improve the performance of the dependent speculation, and in some cases even degraded it (**DepPred_Int**, **DepPred_FP**). This degradation is caused by the fact that the load hit/miss predictor used is too conservative in predicting load misses. The loss in performance for not issuing early for loads that hit is greater than the performance penalty of replaying issued instructions dependent on loads that miss.

Since the misprediction penalty is greater with the sequential scheme than the dependent one, the hit/miss predictor is, in some cases, more useful with the sequential replay scheme (**Seq_Int** vs. **SeqPred_Int**). Nonetheless, sequential load hit speculation obtains only about 50–80% of the performance improvement potential realized by perfect load hit speculation for the integer benchmarks. For the floating point benchmarks, sequential load hit speculation performed poorly, and in some cases worse than the base case of no load hit speculation, even with the predictor. As noted earlier with the perfect predictor, floating point benchmarks tend to have more parallel streams of dependent instructions so the sequential scheme may needlessly replay more instructions that were not dependent on a load miss. Moreover, the total number of instructions replayed increases as pipeline depth (and thereby speculative window size) increases, since there are more instructions in the pipe when it is discovered that a load missed.

The effect of load hit speculation differs significantly between different benchmarks. Figure 3 shows the increase in IPC using dependent load hit speculation, in comparison to no load hit speculation, for a representative sample set of integer and floating point benchmarks. The reasoning behind these variations is the mix of instruction dependencies in the different benchmarks, which allows different issue rates and utilization of the pipeline resources. Overall, as the pipeline becomes deeper, the use of load hit speculation becomes more essential for performance. Choosing a complexity-effective design that can support load hit speculation becomes more important as pipeline depth increases. For the remainder of the paper, we concentrate on load hit speculation with dependent replay of instructions, which we found to be the optimal scheme—and worth the extra complexity compared to sequential or no load speculation.

Figure 3. Performance improvement of using dependent load hit speculation for different benchmarks and varying pipeline depths.

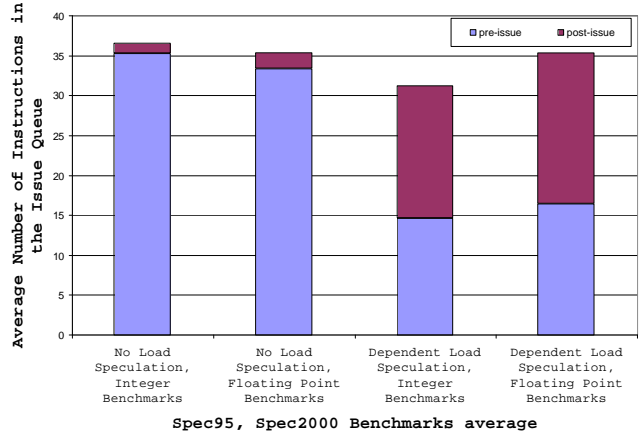


3.2 Effect on Issue Queue

Without load hit speculation, instructions can be removed from the issue queue as soon as they issue and resolve their dependencies. Whereas with load hit speculation, instructions are required to spend more time in the issue queue, since instructions cannot be removed until they reach the writeback stage, and are guaranteed to not be replayed¹. However, instructions also begin issue earlier with load hit speculation, so the overall occupancy of the issue queue may remain the same. In any case, the time instructions spend post-issue (i.e., the time between the point instructions are issued until they are removed from the issue queue) grows with load hit speculation, and with pipeline depth. Figure 4 shows this phenomenon for our deepest simulated pipeline. Without load hit speculation, post-issue instructions on average comprise less than 6% of all instructions residing in the issue queue. When load hit speculation is implemented, on average over 50% of the queue holds post-issue instructions. Figure 5 compares utilization among simulations using load hit speculation. As pipeline depth grows so does the fraction of the issue queue holding post-issue instructions. The percentage of post-issue instructions goes up from about 30% on average to about 55% on average as pipeline depth increases. For deeper pipelines, at some point during program execution most of the issue queue may be occupied by instructions that are waiting to be potentially replayed. This “poorly utilized” issue queue may not have been a concern when pipelines

¹Strictly speaking, we only need to keep the instructions in the issue queue long enough for the hit/miss signal to reach the issue queue. For our implementation, this is effectively the same thing as waiting for them to reach the writeback stage.

Figure 4. Number of pending and post-issue instructions in the issue queue with and without load hit speculation with latency of 7 cycles between the issue and execute of instructions.



were still relatively shallow since the problem is not very pronounced in this case. One solution may be to increase the issue queue size to compensate for the larger fraction of post-issue instruction in order to allow new instructions to enter the issue queue. However, this will only increase the complexity of the issue queue further, particularly in the bid/grant arbitration logic.

As the issue queue size grows, the cost of instruction dependency checking grows, and with it the pressure on the critical paths in the issue queue. The complexity of designing and implementing any issue queue is related to the problem of picking N data ready instructions out of M entries in the issue queue [2]. All ready instructions may bid to issue, but the arbiter must prioritize among these ready instructions in order to determine which of them will be granted an issue slot. Since the ready instructions may reside anywhere in the queue, the grant signals must propagate the length of the issue queue to allow requesting instructions to update their bid request status and allow their dependents to update their ready status. This bid/grant loop that transfers information from the ready instructions to the arbiter and back up to all instructions is a critical path in the queue design.

We may be wasting our resources by searching for bidding instructions in an issue queue consisting largely of instructions waiting to be replayed. Implementing the arbitration logic for a 128-entry queue, for example, may require either additional pipeline stages or a slower clock, as suggested in [1]. By taking these steps, however, we may lose the initial benefits of load hit speculation. By comparison,

Figure 5. Percentage of post-issue instructions of all instructions in the issue queue with dependent load hit speculation for different benchmarks and varying pipeline depths.

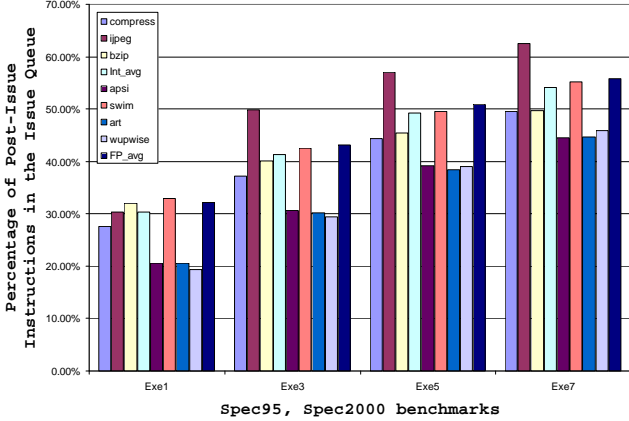
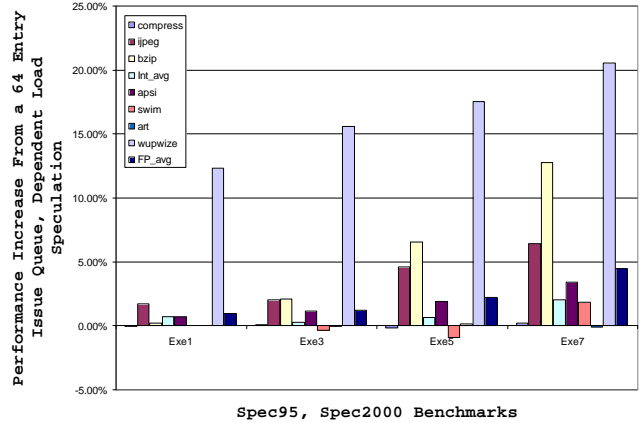


Figure 6. Performance improvement with a 128-entry issue queue, for a sample of benchmarks (using load hit speculation with dependent replay).



sequential load hit speculation may be simpler to implement in hardware relative to dependent load hit speculation, since it does not require searching the issue queue for all post-issue instructions that are dependent on the missed. Instead, it replays all post-issue instructions that were issued after the load. This scheme was used by the Alpha 21264 [7] for integer instructions. The sequential load hit speculation scheme is less likely to require a slower clock or extra cycles, but as shown in Section 3.1, it performs poorly relative to dependent load hit speculation.

4 Impact of Issue Queue Design on Performance

We showed that the combination of load hit speculation and deeper pipelines causes an issue queue utilization problem. Future trends may demand even larger issue queue structures to meet IPC demands. We tried to increase the size of the basic issue queue, in order to justify the need for a better utilized issue queue and show the potential for improvement. Figure 6 shows the performance improvement of a standard 128-entry issue queue over a 64-entry issue queue for a sample of benchmarks. For those benchmarks that are sensitive to the size of the issue queue, the benefit of a larger issue queue increases with pipeline depth. In some points during program execution, at least half of the issue queue may be filled with post-issue instructions. By increasing the size of the issue queue, we are still allowing new instructions to enter the queue. However, part of the performance improvement of the larger queue may also be due to an increase in available ILP, which the larger queue allows.

Meeting the tight timing constraints for a single cycle bid/grant loop will be quite difficult, if not impossible, even for a smaller than 128-entry issue queue. In order to limit the cost of dependency checking, we may choose to implement the bid/grant logic with slower, less complex circuitry. We estimated the effect of a slow, 2 cycle latency bid/grant loop on a 64-entry issue queue by running the base issue queue model with such an implementation. Figure 7 shows the negative effect of increasing this latency. For some of the benchmarks, the performance degraded by as much as 30–60%. On average, it degraded by almost 20% for the integer, and more than 10% for the floating point benchmarks. This leads us to conclude that we cannot afford to have a slower issue queue for a standard unified issue queue. Figure 8 shows similar results on a larger, 128-entry issue queue with 2 cycle latency. Although, the performance is better than a 2 cycle 64-entry queue, it still has a large performance degradation in relation to a standard, single-cycle, 64-entry issue queue. We conclude that even significant increase in the size of the issue queue does not allow the use of a slow select logic.

Another method of limiting the cost of dependency checking, without slowing the select logic, is to limit the size of the issue queue. Figure 9 shows that even reducing the size of the issue queue by as little as 25%, to a 48-entry issue queue, hurts performance for most benchmarks. Benchmarks which can benefit from a larger queue have a performance decrease of up to 20%². As expected,

²Reducing the size of the issue queue may benefit some integer benchmarks (particularly for deeper pipelined processors), since these benchmarks tend to have higher branch mispredictions rates. Restricting the size of the issue queue may inhibit the number of wrong path instructions be-

Figure 7. Performance change of a 64-entry issue queue with a 2 cycle latency compared to a single cycle latency.

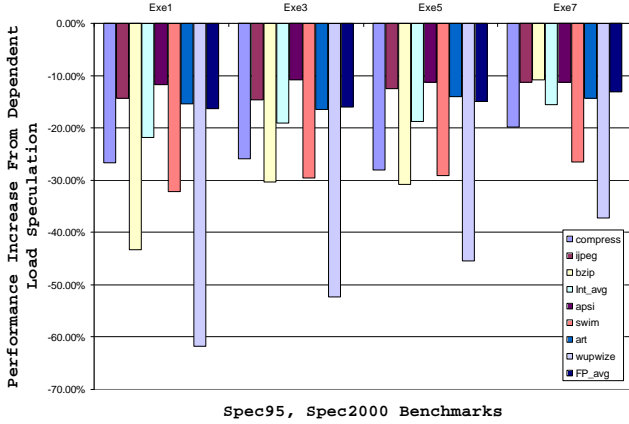
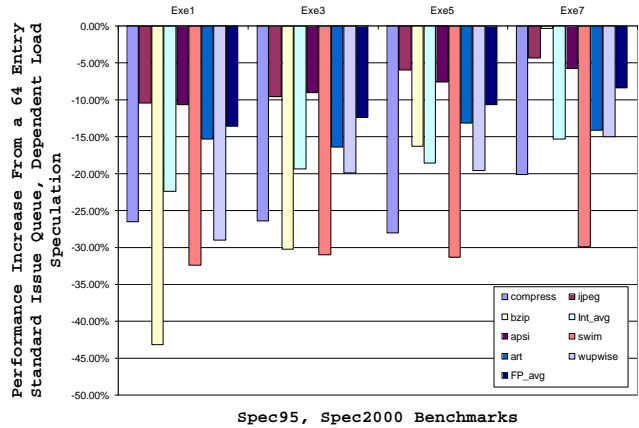


Figure 8. Performance change of a 128-entry issue queue with a 2 cycle latency compared to a 64-entry single cycle latency issue queue.



the benchmarks that benefit from an increase in the size of the issue queue are the ones which suffer the most from a reduction in its size. A 48-entry issue queue is not sufficient, because the smaller issue queue becomes cluttered with post-issue instructions, not allowing new instructions to enter.

5 Dual Issue Queue Scheme

In the previous sections, we showed that the utilization of the issue queue changes as a result of the combination of load hit speculation and deeper pipelines. A larger percentage of the instructions residing in the issue queue have already been issued; these post-issue instructions must remain in the issue queue as long as there is a possibility they will need to be re-issued. What we now propose is a new issue queue design that takes this utilization into account. The goal of this new design is to reduce the complexity of the issue queue without hurting overall performance. The main idea is to move the post-issue instructions out of the issue queue and allow a larger number of pre-issued instructions to reside in the queue, thus increasing the available ILP. We propose to do this with the aid of a separate issue structure that holds these post-issue instructions.

Our new issue queue design consists of two parts: the main issue queue (MIQ), and the replay issue queue (RIQ). The RIQ is effectively a temporary place holder for issued instructions that may need to be replayed. Both queues issue instructions out-of-order; since the instructions are issued out of program order the replay queue must also issue out-of-order.

ing issued and thus prevent useless instructions from wasting processor resources

The pipeline for our new issue queue scheme is shown in Figure 10. The new issue queue structure is shown in gray. Initially, dispatched instructions are placed in the MIQ. The MIQ is searched for ready instructions, and these are given a chance to bid for an issue slot. This part of the issue queue is similar to a standard out-of-order issue queue. Instructions that have already been issued are then moved from the MIQ to the RIQ if there are empty slots available. If the RIQ is full, the issued instructions can remain in the MIQ. After instructions are issued, chances are that they will not be dealt with again, since most load instructions hit in the cache, and their dependent instructions will not be re-issued.

During the issue stage, instructions may be selected for issue either from the RIQ or the MIQ, but not both. To facilitate this, instructions from both queues are allowed to update their request signals every cycle, but only one queue is allowed to bid for issue resources at a time. In our simulation model, the arbitration logic gives priority in selecting instructions to be issued to the RIQ. Only if the RIQ does not have any instructions that are ready to be issued, will the MIQ be searched for ready instructions. An alternative to always giving priority to the RIQ would be to implement a timeout counter for the two queues such that priority would alternate. However, we found no advantage in implementing such a scheme.

The RIQ does not need to be searched for ready instructions every cycle, since the majority of instructions will be issued from the MIQ. Instead, the only time instructions in the RIQ can bid and be granted an issue slot is after a load hit misprediction. In this case, instructions from this queue may be selected for re-issue. Since searching the RIQ for ready instructions is only done after a load hit misprediction, it is not on the critical path of the processor pipeline,

Figure 10. Proposed Dual Issue Queue Scheme.

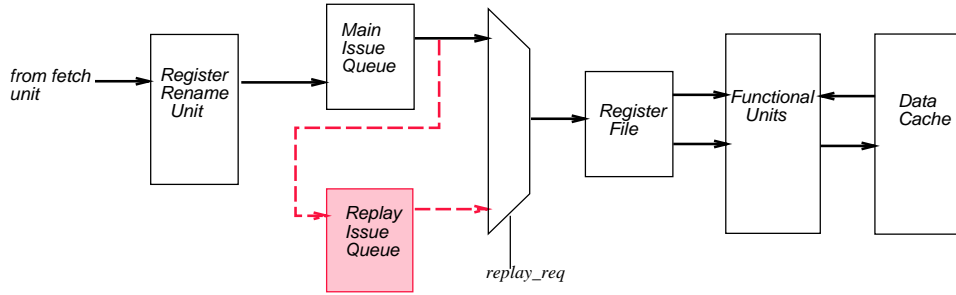


Figure 9. Performance change of a 48-entry standard issue queue (single cycle latency select logic), for a sample of benchmarks (using load hit speculation with dependent replay).

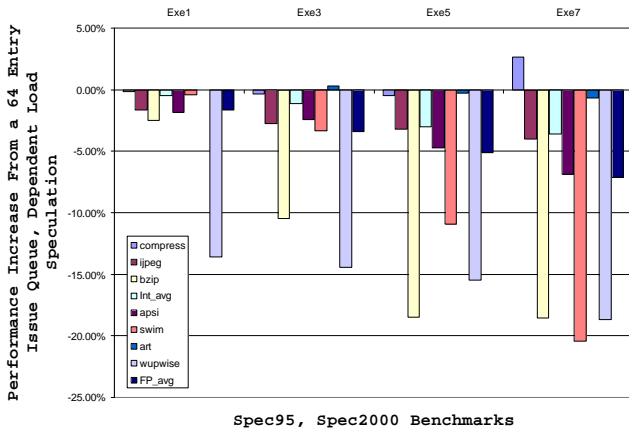
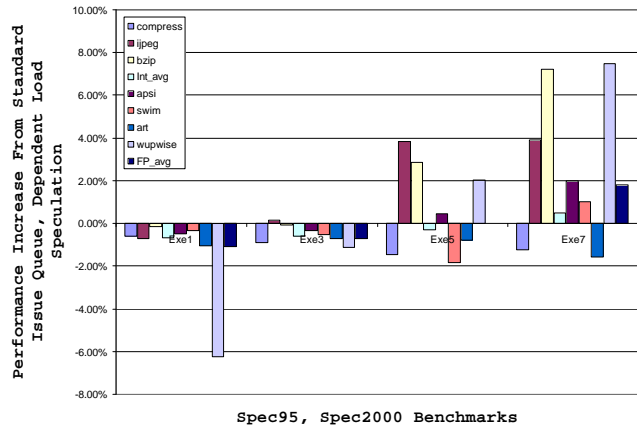


Figure 11. Performance change of the dual issue queue scheme, for a sample of benchmarks (using load hit speculation with dependent replay).



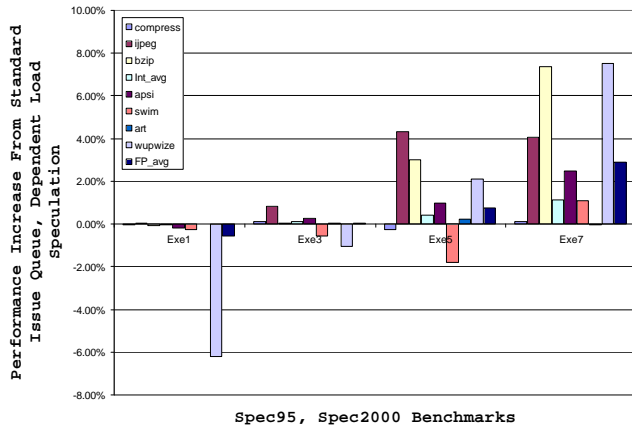
and potentially could be completed in more than one clock cycle. If the RIQ is allowed to operate at a slower rate than the MIQ, its arbitration mechanism may now take two cycles to issue ready instructions. This way, the RIQ can be smaller in physical size, or larger in number of entries, but still less complex than the MIQ.

We found a 48-entry MIQ and a 48-entry RIQ to be an optimal dual issue queue configuration when comparing it to the performance of a 64-entry single cycle issue queue. According to our scheme, the MIQ has a 1-cycle latency arbitration logic, whereas the RIQ has a slower 2-cycle latency arbitration logic. Each cycle, instructions can be selected from only one of the issue queues, with priority given to instructions from the RIQ. Performance results for this scheme compared to a unified single-cycle 64-entry issue queue are shown in Figure 11. Performance does not suffer despite the fact that the main issue queue is smaller; in some cases performance improves by almost 8%. The largest performance degradation is seen for simulations with a 1 cy-

cle issue to execute latency (Exe1), but since our scheme targets deeper pipelines, this is not a concern for us. The largest performance improvements can be seen for benchmarks *bzip* and *wupwise*. These benchmarks are characterized by having a combination of very low d-cache miss rates and few parallel streams of dependent instructions, enabling them to benefit the most from load hit speculation. Notice that overall, we have effectively increased the total number of issue queue entries, but without increasing the complexity of the issue queue. Plus, we can more easily meet the timing requirements of a single-cycle 48-entry issue queue than a 64-entry single-cycle queue, since the bid/grant loop’s delay is limited by the size of the queue.

We would also like to emphasize that having an RIQ with a 2 cycle latency does not compromise performance. Figure 12 shows a similar dual issue queue scheme, only here we allowed both queues to have a single cycle latency. The results for both single-cycle and 2 cycle schemes are very similar, and it can be seen that the cost in terms of performance of a slower RIQ is negligible.

Figure 12. Performance change of the dual queue scheme, with a 1 cycle delay for both queues, for a sample of benchmarks (using load hit speculation with dependent replay).



6 Related Work

Previous work was done on dependency-related issue queue design schemes. Michaud *et al.* [13] proposed adding a preschedule stage before the issue logic, that reorders instructions according to their dependencies, such that they enter the issue buffer in data-flow order. The preschedule stage requires additional logic and data structures to enable the ordering of instructions to be as close to data-flow order as possible, allowing to limit the issue buffer size. Stark *et al.* [17] proposed using pipeline scheduling with speculative wakeup of instructions, in order to allow pipelining of the wakeup and select into two separate stages. In order to speculatively wakeup an instruction, they use the dependency chains of instructions. The speculative wakeup relies on the assumption, that after an instruction is issued, its dependents are to be selected for issue, and using these instructions' latencies, it can be speculated when the next instructions in the dependency chain will be ready for selection. Palacharla *et al.* [14] proposed a complexity-effective issue queue design. Their issue queue scheme consisted of a set of FIFOs, each storing a set of dependent instructions. The instructions are stored in such an order, that the select logic need only search the heads of the FIFOs for instructions ready to issue, thus simplifying the select mechanism. None of the above work discussed the specific effect of load hit misprediction on the dependency-based issue queue structures, or the effects of deeper pipelines.

Recently, Raasch *et al.* [15] described a dynamic issue queue design scheme, based on dependency chains between instructions. The issue queue is divided into segments, according to the expected time until issue of instructions in

the segment. Instructions are selected and moved down in segments, while only instructions residing in the lowest segment can be issued. This scheme specifically deals with the effect of missed loads on the issue of dependent instructions. When a load misses, its dependency chains are stalled in their current issue queue segments. This is an elaborated and complex issue queue design. Borch *et al.* [3] discussed specifically the effects of deeper pipelines on the load loop of the pipeline. They proposed an improved register file structure in order to reduce the loop of load misprediction. Both Sprangle *et al.* [16] and Hartstein *et al.* [10] considered the effects of deepening pipelines on processors performance, with no specific mention of the effects of load hit misprediction.

7 Conclusion

Load hit speculation is an important method in increasing performance and enabling more instruction-level parallelism. We showed that as pipeline depth increases, the use of load speculation increases the percentage of post-issue instructions in the issue queue, limiting the amount of exposed instruction level parallelism. We propose a new complexity-effective issue queue scheme that addresses the utilization concerns without compromising performance. Our dual issue queue allows a larger number of pre-issue instructions to reside in the queue, by dedicating a separate structure to post-issue instructions. In this way, we allow a larger amount of available ILP to be exposed with our dual issue queue scheme compared to a single issue queue scheme even when the main issue queue is smaller. In addition, by making the main issue queue smaller, it can more easily be implemented in a single cycle.

References

- [1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th International Symposium on Computer Architecture*, June 2000.
- [2] R. Iris Bahar and Srilatha Manne. Power and energy reduction via pipeline balancing. In *28th International Symposium on Computer Architecture*, July 2001.
- [3] Eric Borch, Eric Tune, Srilatha Manne, and Joel Emer. Loose loops sink chips. In *8th International Symposium on High-Performance Computer Architecture*, February 2002.
- [4] D. Burger and T. Austin. The simplescalar tool set. In *Version 3.0 Technical Report*, 1999. University of Wisconsin, Madison.
- [5] B. Calder and G. Reinman. A comparative survey of load speculation architectures. In *Journal of Instruction-Level Parallelism*, May 2000.
- [6] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*, June 1998.

- [7] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.
- [8] R. E. Dessler. The alpha 21264 microprocessor. In *IEEE Micro*, March 1999.
- [9] Jeffrey Gee, Mark Hill, Dinoisions Pnevmatikatos, and Alan J. Smith. Cache performance of the spec benchmark suite. In *IEEE Micro*, Vol. 13, Number 4, pp. 17-27, August 1993.
- [10] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. In *29th International Symposium on Computer Architecture*, May 2002.
- [11] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. In *IEEE Computer*, July 2000.
- [12] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. In *Intel Tehcnology Journal*, Q1 2001.
- [13] Pierre Michaud and Andre Sez nec. Data-flow prescheduling for large instruction windows in out-of-order processors. In *7th International Symposium on High-Performance Computer Architecture*, January 2001.
- [14] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *27th International Symposium on Computer Architecture*, June 1997.
- [15] Steven E. Raasch, Nathan L. Binkert, and Steven K. Reinhardt. A scalable instruction queue design using dependence chains. In *29th International Symposium on Computer Architecture*, May 2002.
- [16] Eric Sprangle and Doug Carmean. Increasing processor performance by implementing deeper pipelines. In *29th International Symposium on Computer Architecture*, May 2002.
- [17] Jared Stark, Mary D. Brown, and Yale N. Patt. On pipelining dynamic instruction scheduling logic. In *International Symposium on Microarchitecture*, December 2000.
- [18] A. Yoaz, M. Erez, R. Ronnen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *26th International Symposium on Computer Architecture*, May 1999.