Front-End Policies for Improved Issue Efficiency in SMT Processors[*]

Ali El-Moursy and David H. Albonesi

Department of Electrical and Computer Engineering
University of Rochester
{elmours,albonesi}@ece.rochester.edu

Abstract

The performance and power optimization of dynamic superscalar microprocessors requires striking a careful balance between exploiting parallelism and hardware simplification. Hardware structures which are needlessly complex may exacerbate critical timing paths and dissipate extra power. One such structure requiring careful design is the issue queue. In a Simultaneous Multi-Threading (SMT) processor, it is particularly challenging to achieve issue queue simplification due to the increased utilization of the queue afforded by multi-threading.

In this paper, we propose new front-end policies that reduce the required integer and floating point issue queue sizes in SMT processors. We explore both general policies as well as those directed towards alleviating a particular cause of issue queue inefficiency. For the same level of performance, the most effective policies reduce the issue queue occupancy by 33% for an SMT processor with appropriately-sized issue queue resources.

1 Introduction

The last ten years have witnessed dramatic microprocessor performance gains, in large part due to microarchitectural innovations such as out-of-order superscalar execution and speculative execution. Simultaneous Multi-Threading (SMT) [21] is one such innovation that improves overall instruction throughput via the simultaneous sharing of microprocessor resources among multiple threads. Yet, to maximize performance, one must strike a careful balance between parallelism and clock frequency in designing such processors. Overly complex structures may exacerbate critical timing paths, thereby degrading clock frequency, which may yield a net *decrease* in performance. An additional consideration is power dissipation, which has emerged as a major microprocessor design constraint across-the-board from handheld to server applications. At the high-end, power-related problems include higher packaging costs, noise issues associated with large cycle-to-cycle current swings, room cooling expenses, and power delivery costs. Overly complex hardware structures may impact one or more of these factors, leading to higher costs and/or limited performance.

One structure which has been a major focus of both speed-enhancing and power-saving techniques is the issue queue, which holds a *window* of dispatched instructions until their source operands have been produced and an appropriate functional unit is available. The delay of the issue queue grows quadratically with both the window size and the issue width [14]. As the issue width is increased, so must the window size to effectively exploit this added width. The resulting increase in delay may place the issue queue on the critical path [14]. As a result, recent research has focused on reducing the issue queue delay [9, 10].

The issue queue may also be a major source of power dissipation. For example, in the Alpha 21264 microprocessor, the integer issue queue is the highest power consuming functional block on the chip [23]. The issue queue may also have a high power density [2], which may lead to hot spot problems. Thus, a number of issue queue power-saving techniques have been introduced to attempt to turn off unused or underutilized issue queue entries during application execution. Two such approaches are fine grain clock gating [3] and dynamic adaptation of the issue queue [4, 6, 7, 15]. These approaches rely on the fact that because applications differ in their underlying hardware requirements, there may be phases of execution during which the issue queue may be underutilized. Exploiting such variability is the focus of many of today's issue queue power-saving techniques.

The use of SMT makes achieving the goal of a fast and power-efficient issue queue more challenging for two reasons. First, the window of an SMT processor must hold instructions from several threads in order to achieve its higher throughput. This necessitates growing the issue queue to achieve this larger window. Second, in an SMT processor, the inability of a single thread to utilize the issue queue (for

[*]This work was supported in part by NSF grants CCR–9701915, CCR–9811929, and INT–0117667; by DARPA/ITO under AFRL contract F29601-00-K-0182; by an IBM Faculty Partnership Award; and by external research grants from Intel and DEC/Compaq.

instance, due to an Icache miss) can be made up by filling the queue with instructions from other threads. The result is increased utilization of the issue queue relative to a single-threaded machine, and less opportunity for power savings via dynamic issue queue adaptation or fine-grain clock gating. Indeed, Seng et al. [18] have demonstrated a 22% reduction in the energy per instruction in SMT processors due to this more efficient resource utilization. Thus, new techniques are needed that reduce issue queue resource requirements in SMT processors, thereby reducing both issue queue delay and power dissipation, without unduly impacting performance.

In this paper, we present new front-end policies that reduce the occupancy of the issue queues without compromising performance. Our goals are to increase the number of instructions issued from nearer the head of the queue and to fill the queue with instructions that are most likely to become ready for issue in the near future. At the same time, we desire policies which add minimal complexity compared to the issue queue complexity reduction achieved. We experiment with a number of such policies and achieve a 33% reduction in the issue queue occupancy for a given level of performance compared to a baseline with modestly-sized queues.

The rest of this paper is organized as follows. In the next section, we discuss the reasons for inefficient usage of the issue queue, *i.e.*, *issue queue clog*. We then describe in Section 3 the baseline and proposed front-end policies that we comparatively evaluate. In Section 4, we describe our evaluation methodology, while our results are presented in Section 5. We discuss related work in Section 6, and finally, we conclude and discuss future work in Section 7.

2 Reasons for Issue Queue Clog

Issue queue clog occurs when instructions reside in the queue for many cycles before they are issued. These instructions occupy slots that could be used by instructions which may become ready for issue earlier. This results in a larger queue being required for a given level of performance as well as instructions being issued from deeper in the queue.

One source of issue queue clog is *long latency instructions* that delay the issue of dependent instructions. In most implementations of modern instruction set architectures, integer instructions have relatively short latencies as compared to floating point instructions. Floating point multiply and divide instructions are particularly problematic due to their much longer latencies than floating point add/subtract instructions. However, many microprocessors hold load instructions in the integer issue queue. Loads that miss in the L1 data cache are the greatest source of integer issue queue clog. They also contribute to floating point issue queue clog due to floating point load dependences.

The second source of issue queue clog is *long data dependence chains* that delay the issue of instructions from deep in the chain even if all instructions in the queue have short latencies. Of course, the length of the data dependence chains in the queue is impacted by the first factor, the latency of already-issued instructions at the head of the chain. For instance, a chain of data dependent instructions sourced by a load that misses in the cache may be longer than one sourced by a low-latency instruction (such as a load that hits).

A third source of clog is *contention for functional units* (including the data cache), whereby instructions have their source operands available but cannot issue due to not enough functional units of a given type to handle all ready instructions of that type, and/or the units are not pipelined. This is particularly problematic for workloads primarily composed of operations of a given type, *e.g.*, all integer applications. In many implementations, including the one we model in this paper, the peak issue bandwidth is constrained only by the number of implemented functional units.

We experiment with two overall strategies for reducing issue queue clog that work in conjunction with the ICOUNT fetch scheme of Tullsen [20]. In ICOUNT, priority is assigned to a thread according to the number of instructions it has in the decode, rename, and issue stages (issue queue) of the pipeline. Both of our approaches *fetch gate* threads under particular circumstances, thereby removing them from consideration by the ICOUNT scheme. The first approach uses an approximation of the number of unready instructions that lie in the integer and/or floating point issue queues (without incurring the prohibitive cost of precisely calculating this value) without regard for the reason for their existence. The second approach specifically targets load misses as the primary source of issue queue clog. Our initial design, called *Data Gating,* bears similarity to a previous approach proposed by Tullsen and Brown [22] for performance reasons; however, we propose an extension of this approach called *Predictive Data Gating* that uses load hit prediction to greatly increase its effectiveness. In the next section, we describe various derivatives of these policies, in addition to the baseline front-end policy, in more detail.

3 SMT Front-End Policies

3.1 Baseline Policy

Tullsen [20] explored a variety of SMT fetch policies that assign fetch priority to threads according to various criteria. The best performing policy was determined to be ICOUNT, in which priority is assigned to a thread according to the number of instructions it has in the decode, rename, and issue stages (issue queues) of the pipeline. Threads with the fewest such instructions are given the highest priority for fetch, the rationale being that such threads may be making more forward process than others, to prevent one thread from clogging the issue queue, and to provide a mix of threads in the issue queue to increase parallelism. Two parameters, $numthreads$ and $numinsts$ characterize an ICOUNT scheme (with the designation $ICOUNT.numthreads.numinsts$). The first parameter dictates the maximum number of threads to fetch from each cycle, while the second denotes the maximum number of instructions per thread to fetch.

We also examined the IQPOSN scheme from [20] which attempts to minimize issue queue clog by favoring threads with instructions distributed more towards the tail of the queue. However, we found that IQPOSN provided no significant advantage in either performance or issue queue occupancy over ICOUNT to justify the added complexity of tracking instructions within the issue queue. (ICOUNT, by contrast, only requires per-thread counters that are incremented on fetch and decremented on issue.) Therefore, we choose ICOUNT as the baseline policy against which we benchmark our proposed schemes.

We ran a variety of experiments for different ICOUNT configurations (varying $numthreads$ and $numinsts$) and found ICOUNT2.8 to be the best baseline policy both in terms of performance and issue queue occupancy for our simulation parameters (described in Section 4).

3.2 Proposed Fetch Policies

3.2.1 Unready Count Gating

The first two sources of issue queue clog result in Not Ready instructions (those with one or more source operands not available) occupying the queue for an excessive number of cycles. Unready Count Gating (UCG) attempts to limit the number of Not Ready instructions in the queue for a given thread. The precise implementation of such a policy would count the number of Not Ready instructions in the queue for each thread, but like IQPOSN, this would be prohibitive in terms of hardware complexity. Our simplified UCG implementation operates as shown in Figure 1. A particular thread's Unready Instruction Counter is incremented for each instruction from that thread dispatched into either issue queue in a Not Ready condition. This information is usually obtained from a lookup table (*e.g.*, the Busy Bit Tables in the Mips R10000 [24]) at dispatch time. Each such instruction dispatched is also tagged in the appropriate queue by the setting of the Unready on Dispatch bit, which remains set throughout the lifetime of the instruction in the queue. On instruction issue, the Unready Instruction Counter is decremented for each issued instruction with the Unready on Dispatch bit set. Any thread whose Unready Instruction Counter exceeds a particular threshold is blocked from further fetching.

One potential drawback with our simplified implementation is that it fails to account for Not Ready instructions that become Ready in a queue but do not issue right away due to the aforementioned third source (functional unit contention) of issue queue clog. In actuality, excessive contention for functional units eventually causes data dependent consumer instructions to be dispatched in a Not Ready state. Thus, UCG addresses all three sources of issue queue clog, albeit indirectly in order to simplify the implementation.

We also examine a variation called Floating Point Unready Count Gating (FP_UCG) in which only those instructions dispatched into the floating point issue queue increment the Unready Instruction Counter. The advantage of FP_UCG is that it can be combined with other schemes targeted towards the integer queue, such as those we describe in the next two subsections.

3.2.2 Data Miss Gating

Data Miss Gating (DG) directly attacks the primary source of integer issue queue clog: that due to instructions waiting in the queue for a load miss to be resolved. In this scheme, shown in Figure 2, no fetching is performed for any thread that has more than n L1 data cache misses outstanding. A per-thread counter is incremented on a load data cache miss and decremented when the load commits. A thread is fetch gated whenever its count is greater than n.

3.2.3 Predictive Data Miss Gating

One potential issue with the DG policy is that there is a delay between detection of a fetch gating event (data cache miss) in the Execute stage of the pipeline and the actual fetch gating of the thread. This delay may result in instructions that are dependent on the load being fetched and placed in the queue before the thread is fetch gated, thereby clogging the queue. In Predictive Data Miss Gating (PDG), shown in Figure 3, we attempt to reduce this time dilation by predicting a data cache miss when a load is fetched. The same per-thread counter as DG is implemented, but this is incremented either when a load is predicted to miss, or when predicted to hit but actually misses. The latter can be easily determined using the Load Miss Prediction bit (carried through the pipeline with the load) and the hit/miss outcome. As in DG, the counter decrements when the load commits. A thread whose count exceeds n is fetch gated.

Some recent microprocessors, such as the Alpha 21264 [11], incorporate a load miss predictor to determine whether to speculatively issue consumers of the load. The load miss predictor in the 21264 is comprised of a table of four bit saturating counters whose most significant bit provides the prediction (miss if the bit is zero). The counters are incremented by one on a hit and decremented by two

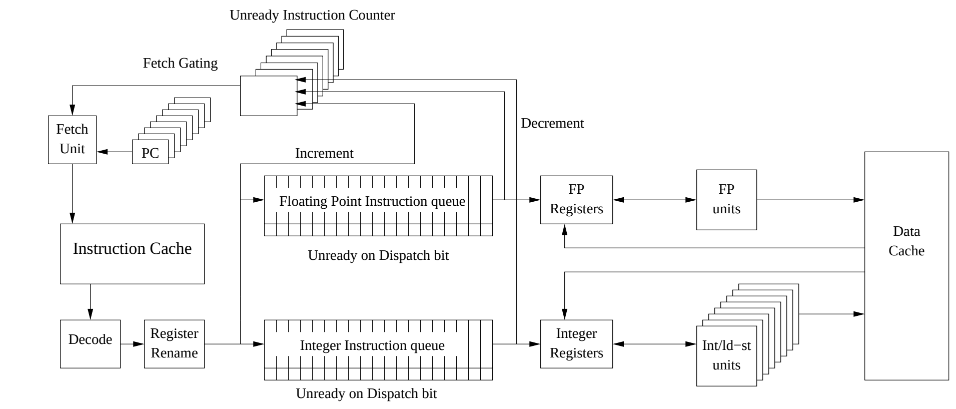

Figure 1. Unready Count Gating (UCG) fetch policy (based on the pipeline in [20]).

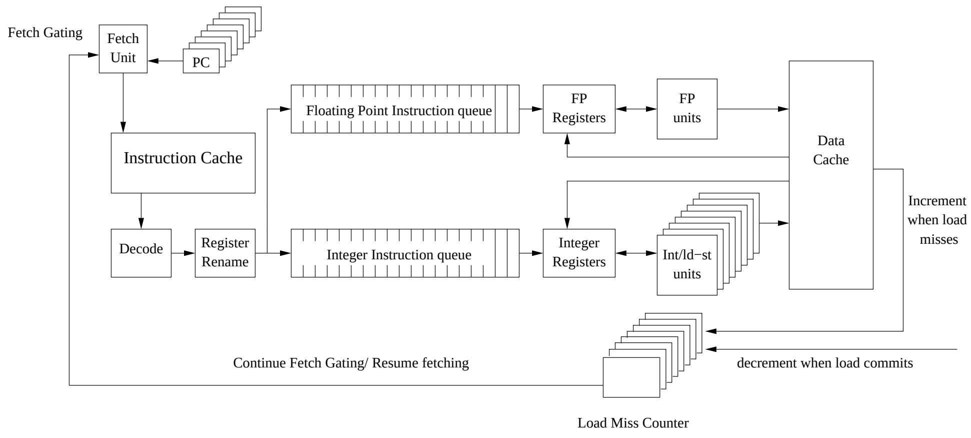
Fetch Gating
Fetch Unit
PC
Floating Point Instruction queue
FP Registers
FP units
Instruction Cache
Data Cache
Increment when load misses
Decode
Register Rename
Integer Instruction queue
Integer Registers
Int/ld−st units
Continue Fetch Gating/ Resume fetching
decrement when load commits
Load Miss Counter

Figure 2. Data Miss Gating (DG) fetch policy.

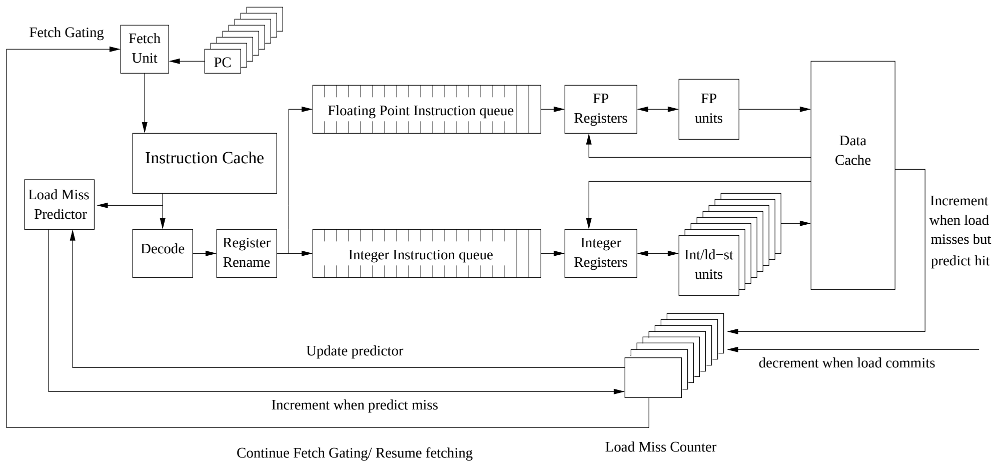
Fetch Gating
Fetch Unit
PC
Floating Point Instruction queue
FP Registers
FP units
Instruction Cache
Data Cache
Load Miss Predictor
Increment when load misses but predict hit
Decode
Register Rename
Integer Instruction queue
Integer Registers
Int/ld−st units
Update predictor
decrement when load commits
Increment when predict miss
Continue Fetch Gating/ Resume fetching
Load Miss Counter

Figure 3. Predictive Data Miss Gating (PDG) fetch policy.

on a miss. By moving the predictor earlier in the pipeline, we can achieve both functions by carrying the Load Miss Prediction bit along with the load as described above.

We explored a variety of predictor options and determined that a 2K-entry table of two bit saturating counters (indexed by the PC of the load) which are cleared on a miss and incremented on a hit, and whose most significant bit determines the prediction, provides an overall hit/miss prediction accuracy of 95%.

3.2.4 PDG Combined with UCG or FP_UCG

Combinations of PDG and UCG or FP_UCG can potentially address multiple causes of both integer and floating point clog. Because PDG is more effective at reducing integer queue clog than UCG, and FP_UCG addresses additional causes of floating point clog over PDG, the combination of these two policies achieves the best results in terms of both performance and issue queue occupancy reduction for a mixed integer and floating point workload of all techniques that we evaluated.

4 Simulation Methodology

We modified the SMT simulator (SMTSIM) developed by Tullsen [19] to implement the new fetch schemes and to gather detailed statistics on the issue queues. The major simulator parameters are given in Table 1. The issue width is equal to the total number of functional units, and issue priority is by instruction age, with older instructions having priority over newer ones. This is essentially modeled in SMTSIM as a compacting issue queue with position-based selection. With fine-grain clock gating applied to such a design, a reduction in issue queue occupancy roughly translates into an equivalent reduction in switching power [3].

For our baseline, we were careful to select appropriate queue sizes so as not to overstate any gains from our techniques. We simulated queue sizes in 8-entry increments using an all-integer workload (described below) to size the integer queue (as this workload is the most performance sensitive in terms of integer queue size) and an all-floating point workload to size the floating point queue (for similar reasons). We increased each queue size until less than a 5% overall performance gain was achieved with an additional increment. Using this approach, we chose 40-entry integer and 40-entry floating point queues for the baseline.

Our workload consists of eight programs from the SPEC2000 integer benchmark suite and eight SPEC2000 floating point programs. We compiled each program with gcc with the -O4 optimization and ran each with the reference input set. From these 16 benchmarks, we created the four, eight-thread workloads shown in Table 2, each of

Parameter	Value
Fetch width	16 instructions
Baseline fetch policy	ICOUNT.2.8 [20]
Pipeline depth	8 stages
Branch Target Buffer	256 entry, 4-way associative
Branch predictor	2K gshare
Branch mispredict penalty	6 cycles
Reorder Buffer entries/thread	512
Architecture registers/thread	32 Int, 32 FP
Rename registers	200 Int, 200 FP
Baseline issue queue entries	40 entry Int/Ld/St, 40 entry FP
Issue queue selection	oldest-first
Issue width	11
Functional units	8 Int (4 handle loads/stores), 3 FP
ICache	64KB, 2-way, 64B line, 8 banks
DCache	64KB, 2-way, 64B line, 8 banks
L2 Cache	512KB, 2-way, 64B line, 8 banks, 10 cycle latency
L3 Cache	4MB, 2-way, 64B line, 20 cycle latency
ITLB size	48 entry
DTLB size	128 entry
Main Memory latency	100 cycles
TLB miss penalty	160 cycles

Table 1. Simulator parameters.

Workload	Benchmarks
all-integer	bzip2, gcc, vpr, gzip, parser, mcf, perlbmk, twolf
all-floating point	applu, lucas, mgrid, art, swim, equake, mesa, galgel
mix 1	applu, lucas, mgrid, art, parser, mcf, perlbmk, twolf
mix 2	bzip2, gcc, vpr, gzip, swim, equake, mesa, galgel

Table 2. Workload mixes.

which consists of 100 million instructions from each benchmark. We fast-forwarded each benchmark according to the guidelines in [17]. As with sizing the baseline issue queue, we used the all-integer workload in analyzing techniques for improving integer issue queue efficiency. Similarly, the all-floating point workload was used in floating point issue queue experiments. In our summary results, we averaged the results of all four workload mixes (3.2 billion instructions in all) in order to simulate the variety of workload mixes encountered in a multi-threaded machine.

As a performance metric, we chose the harmonic mean of the relative instructions per cycle (IPC) ratings of the n threads, calculated as follows:

$$\frac{n}{\sum_n \frac{IPC_{old}}{IPC_{new}}}.$$

This metric penalizes schemes that improve overall performance at the expense of degrading the performance of particular threads, thereby balancing throughput and fairness considerations [12].

For the DG and PDG policies, we fetch gated a thread

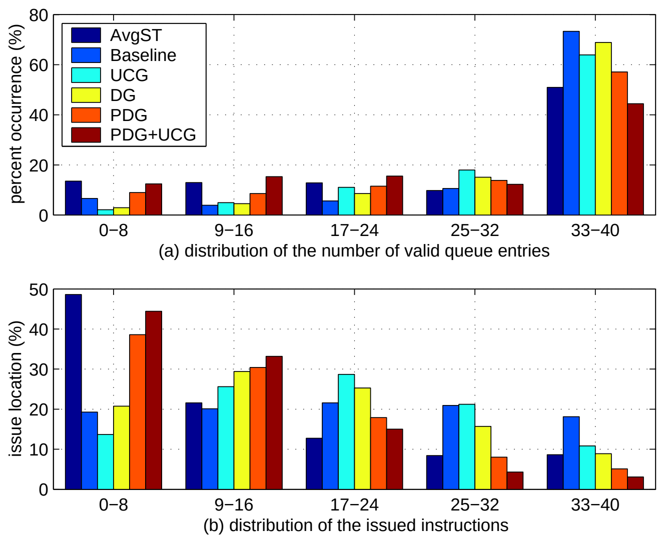

Figure 4. Integer issue queue occupancy and issue distribution for single-threaded, baseline multi-threaded, and proposed policies.

whenever it had an actual or predicted data cache miss, *i.e.*, $n = 0$. For the UCG and FP_UCG schemes, a thread was fetch gated when its Unready Instruction Counter exceeded three and two, respectively. For PDG coupled with FP_UCG, a threshold of three for the Unready Instruction Counter performed best, while for PDG coupled with UCG, a threshold of five was used. A higher threshold is needed in this case to prevent over-gating with the two policies operating simultaneously. We chose these thresholds based on those that performed best for the combination of all four workload mixes. These choices of thresholds are in some cases sub-optimal for the all-integer or all-floating point workloads. This highlights the need to adapt the thresholds dynamically at runtime to fit the workload, which is an area for future work.

5 Results

We first present individual results for the integer and floating point issue queues. In Section 5.3, we present composite results for both queues.

5.1 Integer Issue Queue

Figure 4(a) shows the percentage of cycles that a particular range of issue queue entries were valid, while the (b) part of this figure shows in what range of the issue queue instructions were issued from. AvgST is the average single-threaded result, *i.e.*, averaged over individual runs of each of the eight integer benchmarks. In comparing the single-threaded results with the baseline SMT ICOUNT scheme,

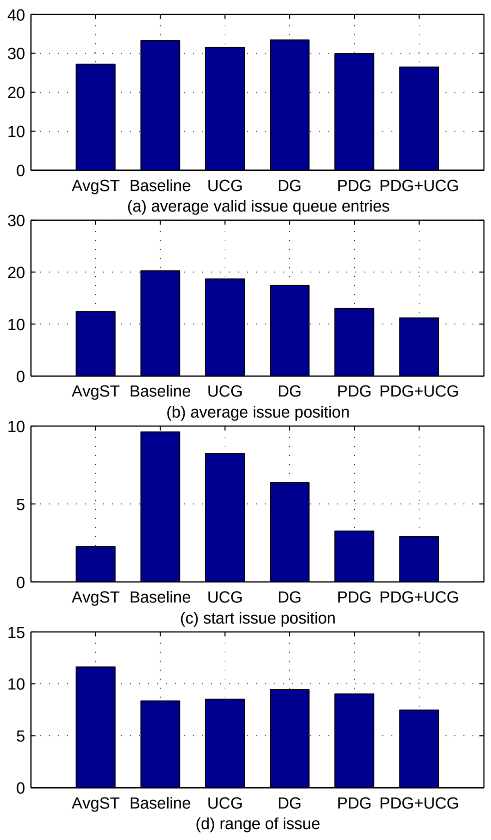

Figure 5. Statistics for single-threaded and multi-threaded policies for the integer issue queue.

we find both a higher overall issue occupancy and a greater number of instructions issued from deeper in the queue with multi-threading. In fact, with ICOUNT, about 75% of the time the queue is nearly fully occupied (33-40 valid entries), while this is the case for only 50% of the time in single-threaded mode. This reflects the greater usage of queue resources in SMT and the ability to find more instructions to issue on average each cycle to achieve greater IPC. The downside, however, is a significant increase in queue occupancy to achieve this higher level of performance.

The distributions for the PDG and PDG+UCG policies closely mirror those of single-threaded mode. This is primarily because these policies prevent the queue from getting clogged with instructions which are unlikely to issue in the near future. For instance, with PDG, over 75% of the instructions are issued from the lower 16 queue entries (exceeding even that of single-threaded mode), whereas roughly 40% of the instructions are issued from these positions with the baseline ICOUNT policy.

Figure 5 provides a variety of statistics for the different policies as applied to the integer issue queue. As shown in Figure 5(a), the enhanced policies, particularly PDG and PDG+UCG, utilize much less of the 40-entry integer issue

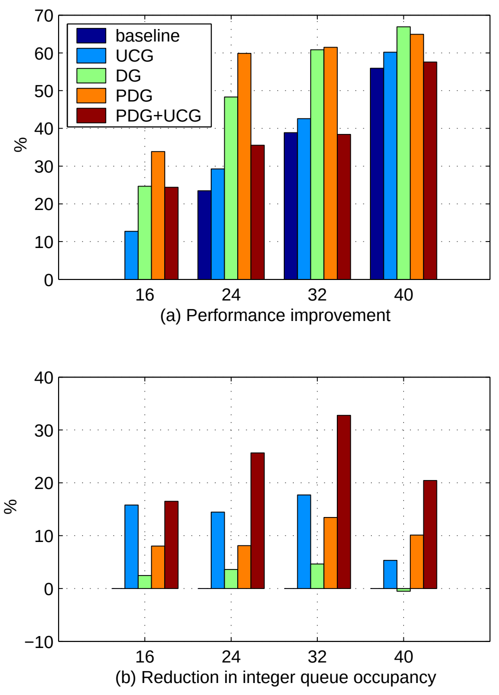

Figure 6. Performance improvement and reduction in integer queue occupancy for multi-threaded policies with varying integer issue queue size.

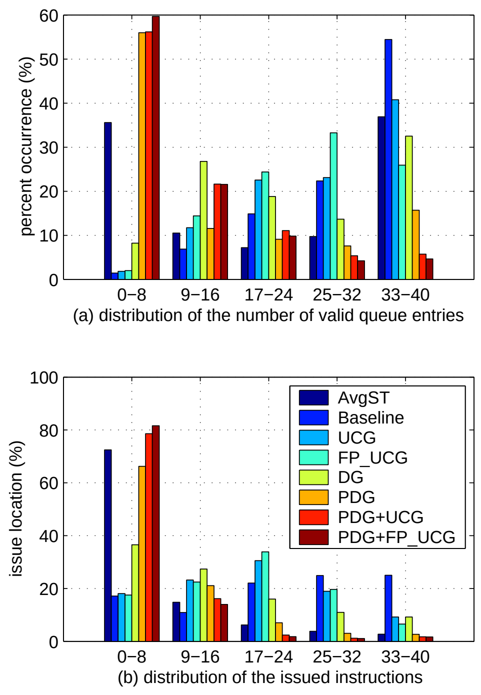

Figure 7. Floating point issue queue occupancy and issue distribution for single-threaded, baseline multi-threaded, and proposed policies.

queue than the baseline ICOUNT policy. As shown in the (b) and (c) parts of this figure, while the average and start issue positions increase dramatically for ICOUNT compared to single-threaded mode, these two enhanced policies reduce these back closer to single-threaded mode levels.

Figure 5(d) shows that the *range of issue*, defined as the number of entries between the first and last instruction in an issue group, remains fairly constant across the multi-threaded schemes. In others words, the number of neighboring instructions needed to be inspected on a cycle to cycle basis is relatively invarient. However, the position of these instructions in the queue varies significantly by policy. On average, the issue queue with PDG behaves comparably to that for a single-threaded workload, yet performance more than doubles.

A salient feature of these enhanced policies is a reduction of the average number of in-flight instructions due to instructions spending less time waiting in the issue queue. This results in a 25-40% reduction in the number of rename registers used as compared to ICOUNT.

Figure 6 plots the overall performance improvement obtained relative to ICOUNT with a 16 entry issue queue for different fetch policies and integer issue queue sizes. The lower part of this figure gives the reduction in the average occupancy of the integer issue queue relative to the baseline with the same number of queue entries.

By all measures (including those in Figures 4 and 5), PDG and PDG+UCG are superior to both UCG and DG. The use of load miss prediction in PDG prevents the queue from being filled with unready instructions as in DG. This dramatically reduces the average start issue position and the average issue position compared to DG, resulting in a significantly greater reduction in occupancy. As mentioned previously, the thresholds in PDG+UCG are tuned to the larger combined workload; for the all-integer workload used for Figure 6, these values yield higher issue queue occupancy savings but worse performance than PDG. In comparison to the baseline, PDG achieves better overall performance with a 24-entry queue than the baseline with a 40-entry queue. This represents a 40% reduction in the required integer issue queue resources to achieve the same level of performance.

5.2 Floating Point Issue Queue

Figures 7, 8, and 9 give similar statistics for the floating point issue queue using the all-floating point workload. With the enhanced fetch policies, floating point issue efficiency improves comparably to that of integer issue. The PDG and combined policies dramatically increase the percentage of time eight or fewer instructions occupy the queue, and the fraction of instructions issued from these positions, even outperforming single-threaded mode on these metrics. With a 40 entry issue queue, PDG achieves a 2-

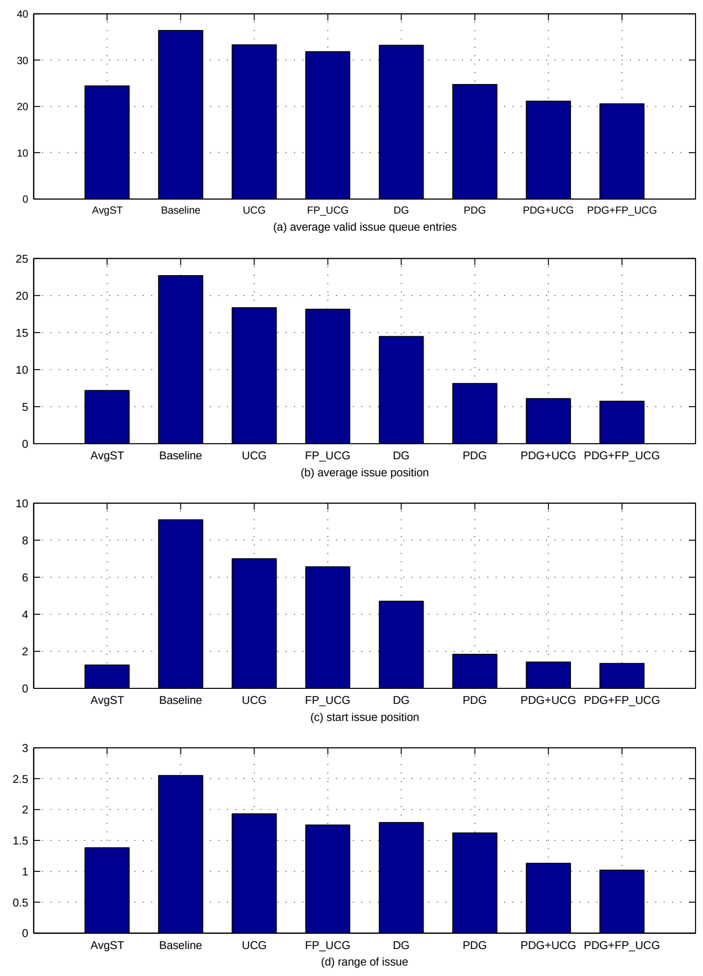

Figure 8. Statistics for single-threaded and multi-threaded policies for the floating point queue.

3X reduction in both the average and start issue positions (Figures 8(b) and (c)), and over a 30% reduction in the occupancy of both queues (Figure 9(b) and (c)) compared to the baseline ICOUNT policy in addition to a performance improvement (Figure 9(a)). The combined policies achieve greater reductions in queue occupancy than PDG alone but with a non-trivial performance degradation, again, due to thresholds that are sub-optimal for this workload.

5.3 Combined Results

Results averaged across all workload mixes in Table 2 are shown in Figure 10 for the three policies that perform best under this workload as well as for DG. Once again, we observe how PDG significantly outperforms DG in all respects due to its ability to perform early and accurate prediction of load misses. On average, PDG+FP_UCG achieves a cumulative 33% reduction in issue queue occupancy while slightly increasing performance. PDG achieves

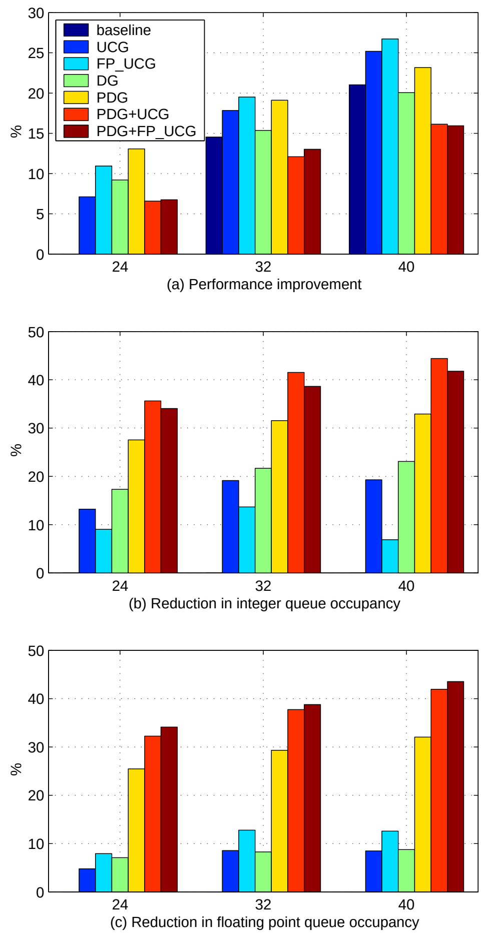

Figure 9. Overall performance, weighted speedup, and floating point issue queue occupancy for different multi-threaded policies with varying floating point issue queue size.

greater performance improvement but at the cost of higher occupancy. In general, the PDG+FP_UCG is more robust than PDG over a range of workloads due to its ability to address other sources of issue queue clog in addition to loads.

6 Related Work

Several techniques for simplifying the issue queue in single-threaded processors have been proposed. Palacharla [13] proposes a dependence-based microarchitecture using multiple smaller queues and grouping dependent instructions in the same queue. Canal [5] develops two schemes for simplifying the queue logic. The first is based on keeping track of the instruction that is the first use of each produced register value. After being decoded, each instruction is dispatched in a different way depending on the availability of its source operands. The second is based on the fact that the latency of most instructions is

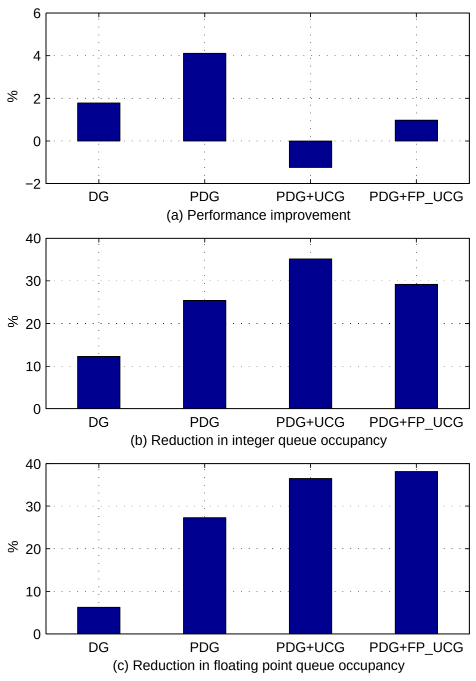

Figure 10. Overall performance improvement, and reduction in integer and floating point issue queue occupancies for different multi-threaded policies using the average of all four workload mixes.

known when they are decoded. These schemes remove the associative look-up and could achieve a shorter cycle time. In [7, 8], Folegnani proposes techniques to dynamically resize the issue queue based on the parallelism in different periods of execution. A 15% reduction in the total power of the processor is achieved with the simplified issue queue. Power reduction of the issue queue via dynamic adaptation is also addressed by Buyuktosunoglu [3, 4] in which a 70% reduction in issue queue power dissipation is achieved with a 3% average performance degradation. Ponomarev [15] and Dropsho [6] expand on this work by resizing multiple structures including the issue queues.

Perhaps the closest work to ours is that by Tullsen and Brown [22] in which fetching is blocked from threads with an outstanding long latency load and instructions from that thread are flushed from the issue queue. Two mechanisms are used to identify long latency loads: an L2 cache miss and a load residing in the load queue beyond a particular number of cycles. With our Data Miss Gating fetch policy, we gate fetching simply based on L1 data cache misses, and we do not add the complexity of flushing instructions. The differences between the approaches are due to the performance focus of [22] and the simplification of the issue queue as the central tenet of our work. In addition, we introduce the Predictive Data Gating and combined PDG and FP_UCG policies that achieve a significant reduction in issue queue utilization over Data Gating as well as higher performance.

Front-end throttling is proposed in [1] as a power reduction technique for single-threaded processors. Three fetch/decode throttling techniques are proposed: Decode/Commit Rate, Dependence-based, and Adaptive. The Decode/Commit policy compares the number of instructions decoded and committed during each cycle to make a throttling decision. The Dependence-based approach counts the dependences among the decoded instructions, while the Adaptive policy combines both methods.

Fetch policies with two priority levels have been investigated in [12, 16]. In [16], the first level priority decision distinguishes between the foreground and the background threads, with ICOUNT and Round Robin schemes prioritizing the threads in each category. However, overall performance degrades using this policy. In contrast, [12] demonstrates combined policies that create a balance between fairness and throughput.

SMT power optimizations have been examined in [18]. The authors demonstrate that an SMT processor with a smaller execution bandwidth can achieve comparable performance to a more aggressive single-threaded processor while consuming less power. They also propose mechanisms to reduce peak power dissipation while still maximizing performance using feedback regarding power dissipation in order to limit processor activity. They also examine the effect of the thread selection algorithm on power and performance and propose favoring less speculative threads over more speculative threads.

In contrast to this previous work, ours is the first to our knowledge that addresses reduction of issue queue complexity in SMT processors via enhanced, yet low complexity, front-end policies.

7 Conclusions and Future Work

The design of aggressive out-of-order superscalar processors requires striking a careful balance between exploiting parallelism and enabling high frequencies. Overly complex hardware threatens to decrease frequency, increase latency, and/or increase power dissipation. The issue queue is one such critical structure where this balance must be achieved to optimize performance and power efficiency. Unfortunately, SMT processors put pressure on increasing the window size in order to hold instructions from multiple threads and to make better use of that window, rendering fine-grain clock gating and adaptive techniques less effective than in single-threaded designs.

We present an approach for reducing the occupancy of both the integer and floating point issue queues without unduly impacting performance. The enhanced front-end policies that we propose both increase the number of instructions issued from near the head of the queue, and prevent the fetching of instructions which are not likely to issue in the near future. The result is a 33% reduction in the occupancy of the issue queues for the same level of performance.

In the future, we plan to evaluate adapting the thresholds dynamically to fit the workload, and to explore the interaction between fetch, dispatch, and scheduling policies on complexity reduction in other areas of SMT processors.

8 Acknowledgements

The authors wish to thank Dean Tullsen for the use of his simulator [19] and his help with our many questions, and the reviewers for their useful comments, especially as related to simulation methodology.

References

[1] A. Baniasadi, A. Moshovos. *Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors*. 5th International Symposium on Low Power Electronics and Design, pp. 16-21, August 2001.

[2] P. Bose, D. Brooks, A. Buyuktosunoglu, P. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, P. Kudva, S. Schuster, J. Smith, V. Srinivasan, V. Zyuban, D. Albonesi, S. Dwarkadas. *Early Stage Definition of LPX: a Low Power Issue-Execute Processor.* Workshop on Power-Aware Computer Systems, in conjunction with HPCA-8, Feb 2002.

[3] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, P. Cook. *Power-Efficient Issue Queue Design.* Power Aware Computing, R. Graybill and R. Melhem (Eds), Kluwer Academic Publishers, Chapter 3, pp. 37-60, 2002.

[4] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, D. Albonesi. *A Circuit Level Implementation of an Adaptive Issue Queue for Power-Aware Microprocessors.* 11th Great Lakes Symposium on VLSI, pp. 73-78, March 2001.

[5] R. Canal, A. Gonzalez. *A Low-Complexity Issue Logic.* 14th International Conference on Supercomputing, pp. 327-335, May 2000.

[6] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, M. Scott *Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power.* 11th International Conference on Parallel Architectures and Compilation Techniques, pp. 141-152, September 2002.

[7] D. Folegnani and A. Gonzalez. *Energy-Effective Issue Logic.* 28th Annual International Symposium on Computer Architecture, pp. 230-239, July 2001.

[8] D. Folegnani and A. Gonzalez. *Reducing Power Consumption of the Issue Logic.* Workshop on Complexity-Effective Design, held in conjunction with ISCA 2000, June 2000.

[9] M. Goshima, K. Nishino, Y. Nakashima, S.-I. Mori, T. Kitamura, S. Tomita. *A high-speed dynamic instruction scheduling scheme for superscalar processors.* 34th International Symposium on Microarchitecture, pp. 225-236, Dec. 2001.

[10] D.S. Henry, B.C. Kuszmaul, G.H. Loh, R. Sami. *Circuits for Wide-Window Superscalar Processors*. 27th International Symposium on Computer Architecture, pp. 236-247, June 2000.

[11] R. Kessler. *The Alpha 21264 microprocessor.* IEEE Micro, 19(2): 24-36, March/April 1999.

[12] K. Luo, J. Gummaraju, M. Franklin. *Balancing thoughput and fairness in SMT processors.* International Symposium on Performance Analysis of Systems and Software, pp. 164-171, Jan. 2001.

[13] S. Palacharla, N. P. Jouppi, J. E. Smith. *Complexity-Effective Superscalar Processors.* 24th International Symposium on Computer Architecture, pp. 206-218, June 1997.

[14] S. Palacharla, N. P. Jouppi, J. E. Smith. *Quantifying the Complexity of Superscalar Processors.* Technical Report CS-TR-96-1328, 1996.

[15] D. Ponomarev, G. Kucuk, K. Ghose. *Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources.* 34th International Symposium on Microarchitecture, pp. 90-101, Dec. 2001.

[16] S. Raasch, S. Reinhardt. *Applications of Thread Prioritization in SMT Processors.* Multithreaded Execution And Execution Workshop (MTEAC), Jan. 1999.

[17] S. Sair, M. Charney. *Memory behavior of the SPEC2000 benchmark suite.* Techical Report, IBM Corporation, October 2000.

[18] J.S. Seng, D.M. Tullsen, G.Z. Cai. *Power-Sensitive Multithreaded Architecture.* International Conference on Computer Design, pp. 199-206, September 2000.

[19] D.M. Tullsen. *Simulation and modeling of a simultaneous multithreading processor*. 22nd Annual Computer Measurement Group Conference, pp. 819-828, December 1996.

[20] D.M. Tullsen, S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, and R.L. Stamm. *Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor*. 23rd Annual International Symposium on Computer Architecture, pp. 191-202, May 1996.

[21] D.M. Tullsen, S.J. Eggers, and H.M. Levy. *Simultaneous multithreading: Maximizing on-chip parallelism*. 22nd Annual International Symposium on Computer Architecture, pp. 392-403, June 1995.

[22] D.M. Tullsen, J.A. Brown. *Handling Long-latency Loads in a Simultaneous Multithreading Processor*. 34th International Symposium on Microarchitecture, pp. 318-327, Dec. 2001.

[23] K. Wilcox and S. Manne. *Alpha Processors: A history of power issues and a look to the future.* Cool Chips Tutorial, in conjunction with Micro-32, 1999.

[24] K. Yeager. *The Mips R10000 Microprocessor*. IEEE Micro, 16(2): 28-41, April 1996.