Fetch Halting on Critical Load Misses[*]

Nikil Mehta,[†]Brian Singer,[‡]R. Iris Bahar
Division of Engineering, Brown University,
Providence, RI 02912
{mehta, bsinger iris}@lems.brown.edu

Michael Leuchtenburg,[§]Richard Weiss
Hampshire College, School of Cognitive
Science, Amherst, MA 01002
michael@slashhome.org,
rweiss@hampshire.edu

ABSTRACT

As the performance gap between processors and memory systems increases, the CPU spends more time stalled waiting for data from main memory. Critical long latency instructions, such as loads that miss to main memory and floating point arithmetic operations, are primarily responsible for these stalls. We present a technique, Fetch Halting, that suspends instruction fetching when the processor is stalled by a critical long latency instruction. This enables us to save power in one of the primary sources of power dissipation, the issue logic. By reducing the occupancy rates in the issue queue and reorder buffer, we save power by disabling a large number of unused queue entries.

In order to characterize critical instructions, our approach combines software profiling and hardware monitoring techniques. Statistical profiling information obtained from sample runs is used to identify critical instructions while hardware cache-miss prediction is used to monitor these instructions. We show that, on average, Fetch Halting can reduce issue queue and reorder buffer occupancy rates by 17.2% and 23.4% respectively, with an average performance loss of only 4.6%.

1. INTRODUCTION

During the execution of a program there are several instances in which the processor is unable to issue new instructions. These generally occur during the execution of certain long latency instructions, such as cache misses to main memory and floating point ALU operations. If a certain long latency instruction has many output dependencies then there is a high probability that the processor will stall until the execution of that instruction is completed.

These stall cycles occur most frequently during accesses to main memory. As processor clock frequencies continue to out pace memory speeds, CPUs are spending more and more cycles stalling idly while waiting for information from main memory. Data prefetching can be effective for minimizing stall time by predicting what data will be needed and preemptively loading that data into the memory caches. However, as memory latency increases, hardware or software based predictors are less adept at hiding the full latency of a cache miss. Mispredictions in highly aggressive prefetching also have negative consequences in that they waste energy via superfluous memory and cache transactions while potentially polluting the caches.

We explore Fetch Halting as a power saving technique by conserving resources during these stall cycles without affecting performance. If the processor is completely stalled then there is no degradation in performance if instruction fetch is halted. In fact, during these stall cycles there is a significant potential for power savings as stalled instructions merely sit in the instruction queue and consume power every cycle via wake-up activity. Disabling these idle entries in the instruction queue during stalls leads to significant power savings by reducing the occupancy rates in the processor's instruction queues. Instruction queues are a significant source of power dissipation; Folegnani *et al.* showed that the issue queue constituted approximately one-fourth of the processor's total power consumption for his assumed architecture [12]. Wakeup activity represented 63% of the power in the issue queue or 16.3% of total power. They then presented a technique to save much of that power by dynamically disabling wakeup for empty issue queue entries. Similarly, [19] and [1] both presented techniques that disabled pipeline queues (issue queue, reorder buffer, and/or load-store queue) according to their occupancy. Fetch Halting takes advantage of these previous approaches by further reducing the occupancy rates of the issue queue and reorder buffer.

To determine when these stall cycles will occur it is necessary to determine which static instructions cause execution to halt. In order for Fetch Halting to be most effective, it is necessary that these instructions be identified as soon as possible during program execution. In this work, we measure the criticality of instructions using a combination of software profiling and hardware monitoring. Software profiling allows us to statically measure the execution penalty of long latency instructions with minimal hardware cost, whereas hardware monitoring allows us to predict dynamically changing load miss behavior. The goal is that this combined monitoring approach will yield the optimum balance of maintained performance and reduced power dissipation.

Fetch Halting is primarily aimed at load instructions that miss to main memory, thereby yielding targets of opportunity with consistently long latencies. By targeting only load misses that are determined to be critical, we show that occupancy rates in the issue queue and reorder buffer can be reduced by up to 31% and 49% respectively while having little impact on performance for most benchmarks. Fetch Halting can then be extended to all long-latency stalling events.

To demonstrate the process of identifying critical static instructions, the software profiling flow is shown at the top of Figure 1. Statistics are first collected for individual instructions from profiling runs of applications using training input sets. Individual instructions are then marked as critical depending on the values of these statistics. Finally, these instructions are annotated within the

[*]This research was supported in part by NSF grant number CCR-0311180 and an equipment grant from Sun Microsystems.

[†]Nikil Mehta is now with the California Institute of Technology, Pasadena, CA.

[‡]Brian Singer is now with QLogic, Aliso Viejo, CA

[§]Michael Leuchtenburg is with the University of Massachusetts, Amherst. This work was completed while at Hampshire College.

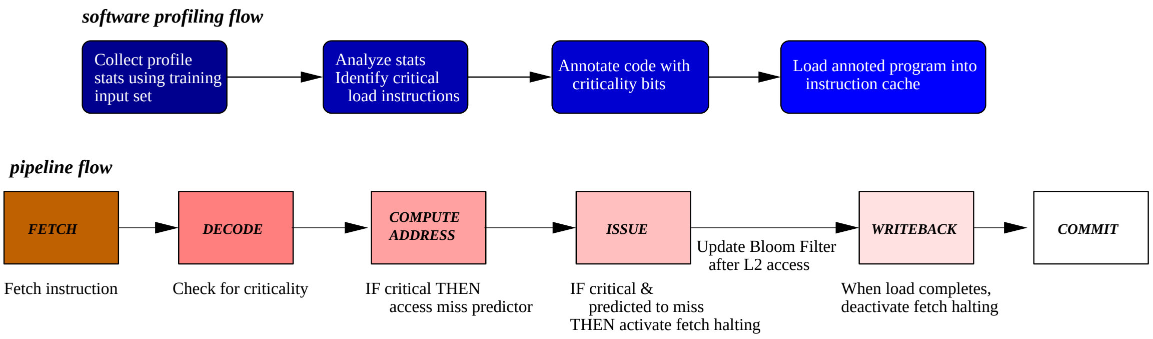
software profiling flow
Collect profile stats using training input set
Analyze stats Identify critical load instructions
Annotate code with criticality bits
Load annoted program into instruction cache
pipeline flow
FETCH
DECODE
COMPUTE ADDRESS
ISSUE
WRITEBACK
COMMIT
Update Bloom Filter after L2 access
Fetch instruction
Check for criticality
IF critical THEN access miss predictor
IF critical & predicted to miss THEN activate fetch halting
When load completes, deactivate fetch halting

Figure 1: Fetch Halting flowchart

program code using annotation, then the program is re-compiled and executed with the reference input set.

The annotated program is executed as shown in the bottom of Figure 1. New instructions are checked for criticality during the decode stage. Once the effective address of a load is computed, the miss predictor is accessed to determine if the load is likely to miss in the L2 cache. If a load instruction is both predicted to miss (by hardware) and marked as critical (through software profiling), then once the load begins accessing the L1 cache, a control signal is sent to the fetch unit such that instruction fetch is suspended starting on the next cycle. Fetching resumes when the load instruction completes or is squashed if it is a wrong-path instruction.

The rest of the paper is organized as follows. Section 2 describes our experimental setup. Section 3 describes our profile-driven technique used to determine a load instruction's criticality. In Section 4 we provide an overview of the Bloom Filter and show how it is used to improve the effectiveness of our criticality predictor. Results are given in Section 5. Finally, prior work is discussed in Section 6 and conclusions and future work are discussed in Section 7.

2. METHODOLOGY

All the experiments were run using a customized simulator derived from the SimpleScalar suite [4]. The simulator was modified to support software profiling of criticality (see Section 3), cache miss prediction (see Section 4), Fetch Halting, and a split issue queue and reorder buffer design. The simulated processor configuration is presented in Table 1.

Table 1: Simulated processor configuration.

Inst. Window	64-entry IQ, 256-entry ROB, 128-entry LSQ
Machine Width	32-wide fetch, 8-wide issue, commit
Fetch Queue	16 instructions
Functional Units	6 ALUs + 2 mult/div 4 FP ALUs + 2 mult/div/sqrt 3 Load/Store units
L1 Icache	16kB 1-way; 16B lines; 1 cycle
L1 Dcache	16kB 1-way; 32B line; 1 cycle
L2 Cache	128kB 8-way; 32B line; 15/20 Icache/Dcache cycles
Memory	64-bit wide; 180 cycles first; 2 inter
Branch Pred.	2k 2-level + 2k bimodal + 2k meta 2 preds/cycle, 3-cycle mispredict penalty

For our simulations we selected a subset of the SPEC CPU2000 integer and floating-point benchmarks [13]. We chose a subset that would demonstrate a range of effectiveness for Fetch Halting; the benchmarks we omitted were either based on Fortran 90 code or showed zero effect from the application of Fetch Halting, due to negligible cache miss rates. [1]

Software profiling was performed on the training input datasets whereas actual results were attained with the reference datasets. Each benchmark was simulated for 100M instructions after fast-forwarding through some number of instructions to avoid simulating startup effects. These fast-forward counts were partially influenced by [20] and [22] and were otherwise obtained by simulating through both the train and ref datasets until some fixed breakpoint in the benchmark. This ensured that both profiling and reference runs were simulating the same part of the program.

We embed the information obtained from software profiling back into the profiled application by annotating the program's compiled assembly instructions and then reassembling these modified source files. Although it is also possible to directly annotate instructions in the program executables, we took advantage of the source code availability of the SPEC benchmarks. One possible shortcoming of this method which will be addressed in the future is that the C library is not annotated, thereby perhaps missing a large potential source of instructions on which to halt.

With respect to an actual implementation rather than simulation, there are a number of profiling tools that can record dynamic information such as load misses and instruction dependency stalls (the time between when an instruction is decoded and when it is ready to execute). For example, the binary re-write tool Spike and the DECC compiler [11] were successfully used to read profile data and optimize the True64 kernel. In addition, Intel has a new set of profiling tools based on Pin [8], which are ISA independent. What we are proposing is even simpler since all that is needed is annotation, which would require taking advantage of unused bits in the instruction word. Aside from that and some counters that will be discussed later, hardware support would not be necessary for profiling and annotation.

3. PROFILING CRITICALITY

To be effective, Fetch Halting must determine if a particular instruction is critical enough that application performance will not be impaired by suspending instruction fetching during its execution. Fetch Halting will only be applied to load instructions that are first profiled as critical and then predicted to miss to main memory. Determining criticality plays a key role in limiting performance loss due to Fetch Halting. Without it, simply halting the fetch unit whenever a load misses to main memory would be severely detrimental to performance.

[1] The single benchmark that does not fall into either of these categories is *ammp*, which because of its extremely low IPC and high cache miss rates took an inordinate time to simulate and was therefore excluded.

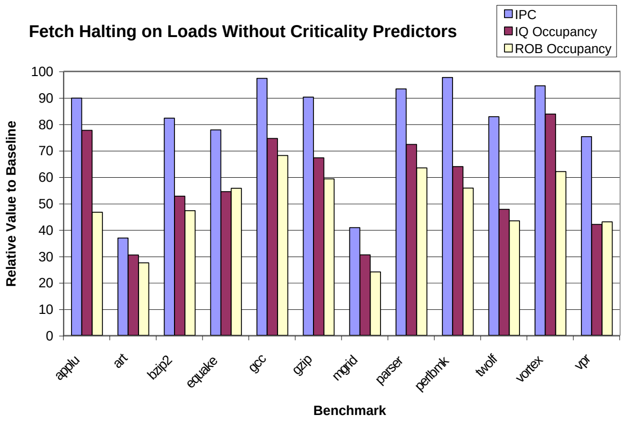

Figure 2: Fetch Halting without considering criticality. Criticality is required to preserve performance with Fetch Halting.

An example of this performance loss is illustrated in Figure 2. For these experiments we assumed perfect knowledge of load miss behavior and only halted the fetch unit when an instruction missed to main memory. The performance drop with this technique averages out to a 20.2% drop in IPC. While we do see overall issue queue and reorder buffer occupancy reductions of 42% and 50.4% respectively, the associated performance loss (as high as 63% in the case of *art*) does not justify the reduction in occupancy.

Fetch Halting uses a software profiling-based criticality predictor. This work considers two profiling heuristics for measuring a load instruction's criticality: *dead cycles* and *fetch-issue count*. The profiling phase collects statistics for these two heuristics for each load instruction. Dead cycles are defined as cycles during which the processor cannot issue any instructions due to unresolved dependencies. The number of dead cycles is counted for each cache miss; this simple count is proportional to the load instruction's criticality. Fetch-issue count is a more specific measurement of performance and criticality. It counts the number of instructions that are fetched and subsequently issued during a miss to memory. A load instruction with a very low fetch-issue count is critical because there is minimal instruction flow when the instruction misses. The actual hardware cost of collecting these statistics would only be trivial counters activated when the processor cannot issue any instructions, and would not impact the overall power savings of our technique.

For each static instruction, the average and standard deviation of dead cycles and fetch-issue count is calculated over all dynamic occurrences of that instruction. Profiling simulations are run using the train dataset in order to quickly obtain representative profiling data that can be later tested using the reference dataset. If an instruction has either a high enough average dead cycle rate or low enough fetch-issue count combined with low standard deviations over a simulation, this instruction is marked as critical. Additionally, in our current implementation Fetch Halting has two levels of criticality: one that will halt the front end for the entire duration of the cache miss (from issue to write-back), and one that will halt for only half that time.

Choosing the thresholds for these metrics that determine if an instruction is critical, half-critical, or non-critical is not an automatic process and involves observing typical dead cycle and fetch-issue counts for a specified processor configuration. Thresholds can be chosen individually per benchmark to tune fetch halting to each application. However, for this work, we chose a more general approach and found that above an average of 165/140 dead cycles or below 70/30 fetch-issue counts worked best for marking critical/half-critical instructions across all benchmarks. Once a critical instruction is determined the actual method of marking an instruction critical, half-critical, or non-critical is performed by using the instruction annotation ability of SimpleScalar. Each instruction uses two annotation bits to indicate criticality.

Several other techniques and metrics are available through this software profiling technique. First, the option of throttling instead of halting is possible, as was used in [24] and [5]. However, we observed that throttling does not have as strong of an impact as halting in reducing queue occupancy. Second, it is also possible to use other metrics for measuring the criticality of load instructions, such as an instruction's average number of dependencies. In our simulations we kept counts of the number of direct and indirect dependencies for each static load instruction; however, we have not been able to fully explore these metrics as useful indicators of criticality yet.

Software profiling can also be extended to long latency instructions such as mispredicted branches or long latency integer and floating point arithmetic operations. For arithmetic instructions the aforementioned metrics are sufficient; however, in the case of branch instructions, several additional metrics are needed to characterize criticality. Most important are measures of branch frequency and misprediction rate. In addition, one may also consider the wrong cycle count (the number of cycles taken down a wrong path for a particular branch), and the wrong path instruction fetch-issue count (the number of instructions fetched and subsequently issued down a wrong path). A combination of these statistics gives a fairly complete picture of what is happening during a specific branch.

Preliminary testing has shown these methods to be promising. The main problem is that some analog to a load miss predictor (such as the Bloom Filter described in Section 4) is needed to improve the dynamic handling of branch instructions. Thus, there is still non-negligible performance loss when using these metrics as criticality predictors. A potential candidate may be the use of a hardware *branch confidence predictor*, as was used in the *Pipeline Gating* approach proposed in [16].

4. CACHE MISS PREDICTION

A simple way of implementing Fetch Halting would be to wait until an L2 miss is detected for a critical instruction and then halt fetching until the data is returned from main memory. However, a more effective approach would halt the fetch in *anticipation* of a miss to main memory. Fetch Halting is most effective if it does not have to wait until the L2 cache miss has been detected in order to be activated.

Software profiling allows us to effectively identify which static loads are critical; however, we still need assistance from hardware to determine if a dynamic occurrence of a load will actually miss to main memory. Peir *et al.* introduced an accurate and power-efficient hardware cache miss predictor in [18]. Fetch Halting uses Peir's Partial-Address Bloom Filter (BF) cache miss predictor (see Figure 3) to improve its effectiveness. This predictor uses the least-significant p bits of the line address to index a small array of bits. Each bit in the array indicates whether the partial address matches any corresponding partial address of a line in the cache. A predictor bit is set when inserting a line into the cache. A bit is reset when evicting a line and if the other ways in the set do not share the same partial address (collision detection). Thus, the predictor has a hardware cost of 2^p bits, and its accuracy increases with the predictor's size (i.e. the number of partial address bits used to index

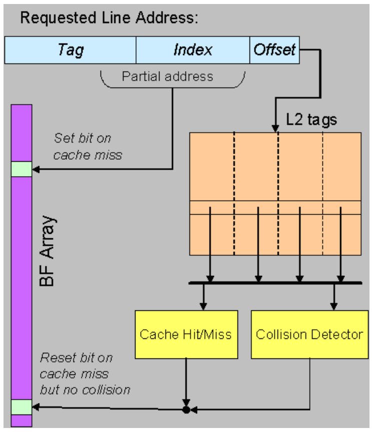

Figure 3: Partial-address Bloom Filter cache miss predictor as introduced in [18].

the predictor). Peir showed that this predictor successfully predicts 97% of all cache misses when it has four times as many entries as are in the cache.

Fetch Halting uses a single BF cache miss predictor for the L2 cache. This should successfully predict nearly every cache miss that results in a main memory access. In these simulations we used a BF predictor with four times the number of entries (8k), which will require 32k-bits or 4kB in hardware to predict a hit in a 128kB L2 cache. While 4kB is not necessarily a negligible amount of hardware, it is still a rather modest overhead — about 3% of the L2 cache size — in exchange for a very high miss prediction accuracy rate. In addition, this is not the only application for which cache miss prediction is useful (see Section 6.4) and it is likely this hardware could be amortized over multiple applications.

5. FETCH HALTING RESULTS

The goal of fetch halting is to reduce the occupancy of the issue queue, which in turn will make the technique of turning off unused queue entries (as proposed in [12] and [19]) more effective at saving power. Specifically, by turning off an unused entry, we can effectively deactivate all logic associated with the wakeup and select logic for that entry. This includes ready, bid, and grant signals for the queue entry — this logic is typically implemented using dynamic circuits, which are very power hungry due to aggressive sizing and high switching activity. This represents a significant portion of issue queue power; however the exact distribution of power consumed by this logic will vary depending on specific design choices. We chose not to use architectural-level power estimation tools such as Wattch [3] because of this variation of power distribution due to design, and also due to Wattch's oversimplification in arbitration logic as discussed in [2] which leads to an underestimation of total chip power savings achieved by techniques targeting issue logic.

The graph in Figure 4 shows the occupancy rates we were able to achieve using either dead cycles or fetch-issue count as our measure of criticality, whichever yielded the best results. Taking the average across 11 benchmarks, we see reductions in issue queue and reorder buffer occupancy rates by 17.2% and 23.4% respectively, with only a 4.6% drop in IPC. In addition, certain benchmarks perform particularly well. Using fetch-issue, *gcc* experienced IQ and ROB occupancy reductions of 20% and 25% with only a 2% drop in IPC. Similarly, we see IQ and ROB occupancy reductions of 29% and 36% in *perlbmk* with a 1% drop in IPC. While both *gcc* and *perlbmk* demonstrated large reductions in occupancy and negligible reductions in performance without using criticality (Figure 2), we see that criticality is needed to avoid severe drops in IPC for other benchmarks while maintaining significant reductions in occupancy. For example, without criticality *twolf* experienced a drop in IPC by 17%, but with criticality IPC only drops by 4.7% while occupancies are reduced by 27% and 23.5%. Most benchmarks can project to significant power savings at a negligible performance loss.

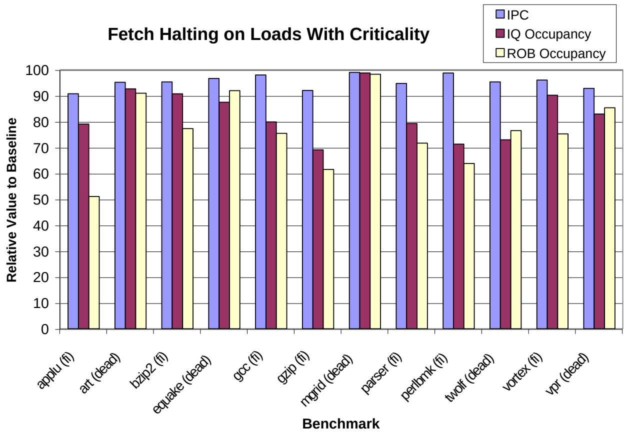

Figure 4: Fetch Halting performance across a range of benchmarks. The words "fi" and "dead" next to each benchmark name indicate whether the fetch-issue or dead cycle metric was used.

A few benchmarks demonstrated less impact. For instance, *mgrid* experienced negligible occupancy reduction. We see similar behavior for *equake* and *parser* For *mgrid* and *equake*, using fetch-issue simply degrades performance too much for Fetch Halting to be useful. We are looking into this issue further to determine if another metric or thresholds customized for a specific benchmark can be found to limit performance loss in these benchmarks while maintaining gains in power reduction. It may also be the case that the characteristic behavior of these benchmarks simply do not allow Fetch Halting to be effective. Consider *parser*, which has an extremely low memory access rate and only a 6.1% L2 miss rate. In comparison, *mgrid* has a 1.1% memory access rate, but while the L2 miss rate is a much higher 53.5%, there are still few overall memory accesses. These rates do not allow many opportunities to halt, which limits the effectiveness of our method.

We believe these numbers can be improved upon, especially considering that we are only acting upon very few static and dynamic instructions (less than 0.1%). Also, although loads are the primary candidate for Fetch Halting, halting on other long latency instructions or branches should improve the performance and power savings of Fetch Halting even more, especially after fine tuning.

A preliminary test of applying Fetch Halting to loads and long latency arithmetic operations is shown in Figure 5. We consider all instructions with a latency of 4 cycles or greater to be "long latency". We see that applying Fetch Halting to long latency arithmetic instructions as well further improves the occupancy rates of benchmarks such as *gcc* and *perlbmk* (an extra 12% and 11% respectively, for both the IQ and ROB) without reducing IPC further (only an additional loss of 1% for both) . Additionally, benchmarks such as *mgrid* that show little to no reduction in occupancy rates (only 1%) for Fetch Halting on loads actually show some reduction in occupancy rates at a small cost in IPC. In this case, *mgrid* achieves an additional reduction of IQ and ROB occupancy of 8%

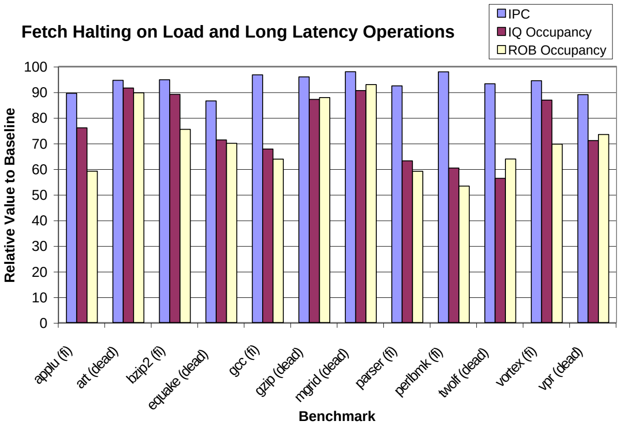

Figure 5: Fetch Halting performance when applied to both loads and long latency arithmetic operations.

and 5% with a cost of only 1% in IPC. On average we found 24.1% and 28.5% reductions in issue queue and reorder buffer occupancies with a 6.5% reduction in IPC. As these are preliminary tests, we expect to further improve upon occupancy reductions while mitigating losses in IPC.

6. PRIOR WORK

6.1 Related Front-End Policies

Chi *et al.* presented methods for combining software and hardware reconfiguration control policies in [7]. They introduced Fetch Halting as a means of controlling the processor's front end to save power. However, their work was incomplete and dependent on an oracle load miss predictor. We have presented a more in-depth analysis of Fetch Halting and added a realistic implementation of a cache miss predictor as well as a sophisticated criticality predictor that yields greater accuracy and energy savings.

Buyuktosunoglu *et al.* introduce a hardware based fetch gating scheme that relies upon hardware characterizations of issue queue occupancy to selectively turn off unused queue entries [5]. Fetch Gating saves power in a similar manner to Fetch Halting; however, we influence the occupancy of the issue queue using indicators determined in software rather than hardware. In addition, their work has no notion of criticality, while Fetch Halting relies on measures of criticality to minimize performance loss.

El-Moursy presented a fetch gating policy similar to Fetch Halting except targeted for simultaneous multi-threading (SMT) processors [9]. Like Fetch Halting, El-Moursy's Predictive Data Miss Gating (PDG) aims to reduce issue queue occupancy to save power and even improve performance due to more efficient use of the shared issue queue by competing threads. PDG performs load-miss prediction by looking up a 2k-entry table of two-bit saturating counters indexed by the PC of the load. Unlike Fetch Halting, PDG does not perform any criticality analysis on the load instructions and is applied more generally to L1 cache misses.

The *Pipeline Gating* approach of Manne *et al.* proposed using branch confidence predictors as a means of inhibiting speculative execution when instructions are highly likely to be fetched along a wrong path, thereby preventing wrong-path instructions from entering the pipeline. As with our work, the ultimate goal of [16] is to reduce energy consumption. However, their approach is guided solely by hardware monitors and specifically targets wrong path execution. Instead, our approach can target any critical instruction that has been identified to cause the pipeline to stall.

Unsal *et al.* proposed a compiler-based approach to reducing energy consumption called Cool-Fetch [24]. Their modified compiler estimated IPC by analyzing dependencies to guide a fetch-throttling mechanism. Although Fetch Halting is also software-based, it takes advantage of profiled execution statistics to better predict critical stalling periods. The Cool-Fetch approach is more limited because it is difficult at compile time to predict criticality (which depends on the interplay of out-of-order instructions in execution) and almost impossible to statically predict load misses. Cool-Fetch is also less effective than Fetch Halting: it only reduces issue queue occupancy by 2.0% on average.

6.2 Restricting the Instruction Window Size

The works of [1], [6], [12], and [19] presented methods that resized the issue queue to dynamically reduce the number of active entries in the instruction window according to the processor's and application's needs. Fetch Halting shares the goal of reducing the instruction window size. However, Fetch Halting accomplishes this task by proactively limiting instruction flow rather than by directly constraining the queue sizes. In a sense, our approach could be used in combination with these techniques to make resizing more effective.

Karkhanis *et al.* also studied directly restricting the number of in-flight instructions, except with a coarser granularity (100k-instruction sampling intervals) hardware scheme [14]. They used a tuning method that alternated between the most extreme configurations to determine performance boundaries, and would then make finer adjustments to the configuration until the sampled performance fell within some threshold of the maximum attained performance. Our approach eliminates this need for sampling intervals and provides for a finer level of granularity, which in turn allows us to better tune performance and power.

6.3 Criticality

Tune *et al*. proposed hardware critical path prediction in [23]. Their approach utilizes a large lookup table of 6-bit saturating counters indexed by PC. The counters are incremented when an instruction has been identified as critical according to one of several heuristics. Their most effective heuristics tracked the oldest unissued instructions or the oldest uncommitted instructions. Seng *et al.* showed that these hardware criticality predictors were effective in power-saving schemes that routed instructions through either in-order or out-of-order issue queues and through fast or slow execution units [21].

Fields *et al*. presented an accurate dynamic dependence-graph-based model of critical paths [10]. This weighted graph model reflects the execution of each instruction including the dispatch, execution, and commit stages. Software profiling is used to build this graph, and the critical path is the longest weighted path through the graph. From this model, they designed a hardware predictor that uses a token-passing algorithm to trace the critical path without actually building the graph model. While this approach is more accurate than [23], it also adds another level of complexity that our approach does not require.

A *down-FSM* monitor was employed in [15] to track instruction-issue rate for a small sample period after an L2 miss is detected. An issue rate below a certain threshold triggers the supply voltage to be scaled down. Since they must wait until the L2 miss is actually detected, some lost opportunity in saving power may occur.

6.4 Cache Miss Prediction

Cache miss prediction has proven to be a source of many potential applications in addition to prefetching. The hardware cache miss predictor used in this paper was introduced in [18], which used the predictor to accurately schedule instructions dependent on load instructions. The authors were able to achieve 99.7% of the performance potential of the oracle processor with perfect scheduling.

Memik, *et al.* presented five hardware cache miss predictors and used them to bypass cache layers in a simulated processor with five cache levels [17]. Like the Bloom Filter predictor, Memik's cache miss predictors also focuses on the line address of the cache entries. His techniques include storing evicted addresses, address hashes, or the partial address bits in a table. Applying these cache miss prediction techniques yielded an average application performance increase of 5.4% and average cache power savings of 3.8%.

7. CONCLUSION AND FUTURE WORK

Software profiling is ultimately a tradeoff between additional time and effort for the software developer and increased hardware complexity and cost. We have shown that it is an effective and practical approach for controlling power-efficient reconfigurations. It is also effective because of the opportunity to optimally tune reconfigurations on a per-application basis without adding to the hardware budget.

Dynamic reconfiguration has traditionally focused more on the hardware approach, and for good reason. Utilizing a hardware approach is the best way to take advantage of identifying dynamic behavior that is unpredictable prior to runtime (e.g. cache miss prediction). The downside to the hardware approach is the additional hardware cost for control logic and tables.

Combining the complementary strengths of software and hardware monitoring approaches can lead to a best-of-both-worlds solution. This paper has shown that such a combination can yield very promising results for Fetch Halting. While focusing on L2 cache misses to main memory, which only constitute less than 0.1% of total static and dynamic instructions on average, Fetch Halting can efficiently reduce the average instruction queue and reorder buffer sizes by up to 31% and 49% respectively (17.2% and 23.4% on average) with very little performance loss.

While Fetch Halting is effective when targeting L2 cache misses to main memory, since it only targets a very small percentage of total static and dynamic instructions there is still potential in targeting and tuning for other critical instructions such as long latency arithmetic and branch instructions. We have seen the performance impact of not using criticality metrics on both performance and issue queue and reorder buffer occupancy rates, and in expanding our focus we are using similar metrics to determine criticality. Thus far our results have been promising and after subsequent tuning we expect to achieve similar power savings and negligible performance losses that parallel the success of targeting L2 misses.

Acknowledgment

The authors would like to thank Eric Chi, A. Michael Salem, and Emiliano Bergamaschini for their work with the initial profiling code and simulations.

8. REFERENCES

[1] J. Abella and A. Gonzalez. On reducgin register pressure and energy in multiple-banked register files. In *International Comference on Computer Design*, pages 14–20, San Jose, CA, October 2003.

[2] Y. Bai and R. I. Bahar. A low power in-order/out-of-order issue queue. *ACM Transactions on Architecture and Code Optimization*, 1(2), June 2004.

[3] D. Brooks, V. Tiwari, , and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of International Symposium on Computer Architecture*, Vancouver, British Columbia, Canada, June 2000.

[4] D. Burger and T. Austin. The simplescalar tool set. Technical report, University of Wisconsin, Madison, 1999. Version 3.0.

[5] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose. Energy efficient co-adaptive instruction fetch and issue. In *30th International Symposium on Computer Architecture (ISCA '03*, San Diego, CA, June 2003.

[6] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power at high performance. In *Workshop on Power-Aware Computer Systems*, Cambridge, MA, November 2000.

[7] E. Chi, M. Salem, R. Weiss, and R. I. Bahar. Combining software and hardware monitoring for improved power and performance tuning. In *7th Annual Workshop on Interaction Between Compilers and Computer Architecture (INTERACT-7)*, Anaheim, CA, February 2003.

[8] R. Cohn and R. Muth. Pin user guide. http://rogue.colorado.edu/Pin/docs/pin-2.0/html/.

[9] A. El-Moursy and D. H. Albonesi. Front-end policies for improved issue efficiency in smt processors. In *Ninth International Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 2003.

[10] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *28th International Symposium on Computer Architecture (ISCA '01*, Göteborg, Sweden, July 2001.

[11] R. Flower, C.-K. Luk, R. Muth, H. Patil, J. Shakshober, R. Cohn, and P. G. Lowney. Kernel optimization and prefetches with the spike executable optimizer. In *Proceeding of 4th Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[12] D. Folegnani and A. González. Energy-effective issue logic. In *Proceedings of International Symposium on Computer Architecture*, Göteborg, Sweden, July 2001.

[13] J. L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[14] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *International Symposium on Low Power Electronics and Design (ISLPED*, August 2002.

[15] H. Li, C. Cher, T. Vijaykumar, and K. Roy. Vsv: L2-miss-driven variable supply-voltage scaling for low power. In *Proceedings of International Symposium on Microarchitecture*, San Diego, CA, December 2003.

[16] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *Proceedings of International Symposium on Computer Architecture*, Barcelona, Spain, June 1998.

[17] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Ninth International Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 2003.

[18] J.-K. Peir, S.-C. Lai, and S.-L. Lu. Bloom filtering cache misses for accurate data speculation and prefetching. In *Proceedings of International Conference on Supercomputing*, June 2002.

[19] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings of International Symposium on Microarchitecture*, Austin, TX, December 2001.

[20] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical Report RC-21852, IBM Thomas J. Watson Research Center, October 2000.

[21] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *Proceedings of International Symposium on Microarchitecture*, Austin, TX, December 2001.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[23] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

[24] O. S. Unsal, I. Koren, C. M. Krishnan, and C. A. Moritz. Cool-fetch: Compiler-enabled power-aware fetch throttling. In *Computer Architecture News*, April 2002.