Combining Software and Hardware Monitoring for Improved Power and Performance Tuning

Eric Chi, A. Michael Salem, R. Iris Bahar
Brown University, Division of Engineering
Providence, RI 02912

Richard Weiss
Hampshire College, School of Cognitive Science
Amherst, MA 01002

Abstract

By anticipating when resources will be idle, it is possible to reconfigure the hardware to reduce power consumption without significantly reducing performance. This requires predicting what the resource requirements will be for an application. In the past, researchers have taken one of two approaches: design hardware monitors that can measure recent performance, or profile the application to determine the most likely behavior for each block of code. This paper explores a third option which is to combine hardware monitoring with software profiling to achieve lower power utilization than either method alone. We demonstrate the potential for this approach in two ways. First, we compare hardware monitoring and software profiling of IPC for code blocks and show that they capture different information. By combining them, we can control issue width and ALU usage more effectively to save more power. Second, we show that anticipating stalls due to critical load misses in the L2 cache can enable fetch halting. However, hardware monitoring and software profiling must be used together to effectively predict misses and criticality of loads.

1 Introduction

Although the main driving force in high-end microprocessor design is performance, power consumption is of great concern. These general-purpose processors include many complex architectural features in order to achieve high performance for the broadest set of applications. However, applications vary widely in their degree of instruction-level parallelism (ILP) and execution behavior. As a result, the datapath resources required to implement these complex features may not be optimally utilized by all applications; however, some power will be dissipated by these resources regardless of utilization.

To better address power concerns, a good design strategy should be flexible enough to dynamically reconfigure available resources according to the program's needs. However, in order to determine when and how to reconfigure the machine, some appropriate means of monitoring the application is needed. Current techniques tend to be based on simple hardware monitoring or software profiling, but not both. While either technique may be effective in determining a good configuration, both have drawbacks.

Hardware monitoring itself can encompass a variety of features. Perhaps the simplest technique one can use is to monitor throughput in instructions per cycle (IPC) and implement a control strategy based on monitoring windows and thresholds. The underlying assumption behind this is that current measurements are indicative of future behavior. For example, low IPC might suggest that the instruction queue size can be reduced along with the fetch rate and issue width since the pipeline already has more instructions than it can execute. However, hardware approaches cannot sample multiple configurations easily to determine the best option.

In practice, hardware monitoring must first establish a pattern or determine a specific behavior (such as low IPC) for some interval of execution and then *react* to this new behavior after it has already been established. The fact that the hardware must first detect a change in behavior means that there will always be a lag in optimal machine configuration at which point it may have lost a key window of opportunity. Although smaller sampling windows may help to reduce this lag, it may also lead to spurious changes in machine configuration. Furthermore, the optimal sampling window size may vary from application to application or even inside an application.

Software profiling can identify specific behavior for a short sample run of a program and then annotate instructions appropriately to identify this behavior for future executions of this code. In this way, software profiling allows processor resources to be adjusted in *anticipation* of changing processor requirements. Since annotations are set for static rather than dynamic instructions, only instructions or subroutines with very deterministic behavior should be used to trigger a reconfiguration. This may lead to lost opportu-

nities in saving power and/or performance losses if behavior from the sample run does not closely match the actual run.

In this paper we aim to better exploit the *anticipation principle*, in order to better configure a processor for performance and energy savings. That is, if it is known in advance what the processing requirements will be, processor resources can be shut down and re-enabled more effectively. For example, knowing in advance that a section of executing code has little instruction-level parallelism (ILP) and thereby low IPC, we can save power by instructing the processor to disable the instruction fetch unit, reduce the size of the instruction queue, and disable some of the instruction issue arbiters. Without the anticipation principle, the instruction queue will eventually fill up and shut down the fetch unit, at which point it is not possible to reduce the size of the issue queue until it is sufficiently drained. Furthermore, the instructions in the queue have a high degree of dependence, so that they issue serially, and it takes a long time to drain the instruction queue. Several researchers have noted that anticipatory fetch throttling can actually improve performance as well as power consumption by avoiding fetching instructions down a mispredicted branch.

Because cache misses result in many cycles of stalling for main memory, a critical load instruction may lead to consistently long periods of slow execution. Successful prediction of these cache miss events can provide excellent opportunities for applying the anticipation principle in an effort to save power.

The main contribution of this work is in demonstrating how hardware and software profiling techniques may be used together to strengthen the control policies in power-saving machine reconfigurations. This will lead to more optimal configurations, better energy savings, and higher overall performance.

The rest of the paper is organized as follows. Section 2 provides background and discusses related work. Section 3 presents our methods for reconfiguring the processor as well as the different techniques for hardware and software profiling to drive the reconfiguration. Section 4 describes our processor model and simulation tools. Results are provided in section 5. Section 6 offers conclusions.

2 Background and Prior Work

Several approaches have been explored for dynamically monitoring program behavior and adjusting available resources to better match program requirements. In [15], Maro *et al.* proposed selectively disabling part of the integer and/or floating point pipelines during runtime to save power. Feedback from hardware performance monitors was used to guide the reconfiguration. A similar approach, called pipeline balancing (PLB) was used in [2] where issue width was varied to allow disabling of a cluster of functional units. Other works include selectively gating off parts of the pipeline when branches have a high probability of mispredicting [14], dynamically reducing the number of active entries in the instruction window according to processor needs in order to save power [5, 8, 16], using dynamic voltage scaling on selected regions of the processor [12, 19], and modifying cache size and associativity [3]. All the above mentioned approaches use hardware monitoring to constantly assess the program requirements on the assumption that current behavior is a good predictor of short term future behavior.

Profiling has also been used for critical path analysis. In [21], the authors used hardware profiling to analyze the characteristics of critical and non-critical instructions and designed a static mixed in-order and out-of-order issue queue to reduce power consumption in processors. Fields *et. al* proposed a similar approach in [7], but with an accurate slack-based predictor. Finally, Semeraro *et. al* proposed dividing the processor into multiple clock domains where the clock frequency of each domain could be separately tuned to save power [20]. Using an off-line analysis tool, they constructed a directed acyclic graph (DAG) using trace information collected over 50K cycle intervals, and from the DAGs, determined which instructions could be run at a slower frequency.

There have been more complex hardware monitoring schemes, such as the one studied by Dhodapkar and Smith [6]. They addressed three problems for multi-configuration caches: detecting a change in the working set, identifying the new working set, and using the working set size to select the cache configuration. Working set signatures were stored in a small table. The use of such a table makes this approach more powerful than the previous monitoring schemes. The problem of the latency between a phase change and its detection seems not to have been an issue, since cache reconfiguration is not something that would be done more frequently than once in 10,000 cycles.

An approach that combined hardware monitoring with very simple information from the operating system was proposed in [9]. In this work, the micro-architecture was adjusted to meet the desired performance indicated by the operating system. The processor was changed from out-of-order to in-order according to the program's needs. The reconfiguration is driven by the operating system rather than the hardware and as such may only allow for course grain adjustments.

Other software based approaches that dynamically reconfiguring processors resources include the work of [11] and [23]. In [11], the most frequently executed functions, or modules, are identified and then statistics for these modules are collected during profiling runs of the code. A different profiling run is required for every configuration that may be applied (e.g., enabling a filter cache [13], reduc-

ing ALUs, or using a phased cache [10]). Once profiling is completed, they use a selection algorithm to choose the best overall configuration for each module. While this software profiling approach can lead to good configuration selection, it requires a significant profiling overhead since separate runs are required for each low-power configuration considered. In the work of [23] the authors propose a compiler based approach to reduce energy consumption in the processor. The compiler estimates instructions per clock (IPC) using dependence-testing-based analysis to guide a fetch-throttling mechanism. The idea is that by preventing the fetch unit from running far ahead of the execution unit with IPC is low, wrong-path instructions may be prevented from entering the pipeline. We also propose a software-based scheme to guide fetch-throttling; however, by collecting statistics at the profiling level, we are also able to better predict dependencies, specifically for non-deterministic latencies such as for load misses.

3 Implementation

The goal of our approach is to combine hardware and software profiling to better react to program changes. This will allow us to react as quickly as possible to strongly deterministic changes while at the same time allow the hardware to handle harder-to-predict cases with the help of hints from software profiling. Combining the two also may allow us to select more than one hardware resource to reconfigure, thereby potentially saving more power.

3.1 Low-Power Configurations

To achieve the highest possible energy savings for a mix of applications, we would like to have available a variety of configurations for various hardware components in the processor. We consider two configurations in our initial experimental design:

Reducing Issue Width and ALUs: A drop in IPC may indicate that the issue width and number of functional units may be reduced without impacting performance. Our processor configuration consists of two clusters of integer functional units. We can save power by disabling one of these clusters and reducing issue width in half (and thereby disabling those issue arbiters).

Fetch Halting: If the processor is stalled for an extended period of time due to a long latency cache miss, we gate off fetching until the event has resolved. This may prevent wrong path instructions from entering the pipeline due to a mispredicted branch and may reduce occupancy rates in the fetch and issue queues thereby allowing some portion of these structures to be disabled.

3.2 Software Profiling

In its most general form, software profiling can provide information to the compiler for a recompilation phase. Profiling may be either hardware-based, continuous sampling (DCPI) [1] or involve adding instrumentation code to the program so that the program actively collects statistics. In any case, the resulting data can be fed back into the compiler/assembler together with the source code or fed into a post-link optimizer. In our case, it is not necessary to refer back to the high-level source code as it is simpler to just add annotations to the assembly code produced by the compiler. There are a few different post-link optimizers that have been developed to incorporate profile data in this way. In practice, post-link optimizers are less complex and easier to modify than highly optimizing compilers. In addition, post-link optimizers have the advantage that one can optimize programs without having the source code and that optimizations are automatically tuned to the actual hardware even if the compiler has no model of such an architecture.

Since we are conducting our experiments using SimpleScalar, profiling is performed by collecting statistics such as local IPC and L2 cache misses in the simulator. From the profiling data we can calculate mean and variance in IPC and L2 cache miss patterns and criticality and use this in implementing our reconfiguration control policies. For example, if the average IPC of a block of code is below a certain threshold value, we can annotate the instructions in the block to direct the CPU to reduce issue width mode and disable functional units.

3.3 Hardware Monitoring

As with software profiling, we can also use hardware performance monitors to keep track of various statistics while a program is executing. These statistics are gathered during a fixed-sized sample window. At the end of each sample period, we determine whether to reconfigure the processor, or leave it in the current configuration. In this way, reconfiguration takes place at most once within a single window. We empirically chose our sample window size to be 256 cycles such that it is large enough to obtain meaningful statistics over a reasonable span of time, but not too large to remain in an inappropriate configuration. Previous work using hardware based profiling proposed monitoring several different statistics such as IPC, floating point IPC, resource utilization, and instruction dependencies [2, 5, 8, 15, 16, 18, 21]. Although all these statistics may be useful to varying degrees in determining how to optimally configure a processor, IPC has been shown to be a particularly useful statistic to monitor. Therefore, for now we will limit our hardware profiling analysis to consider only IPC variation.

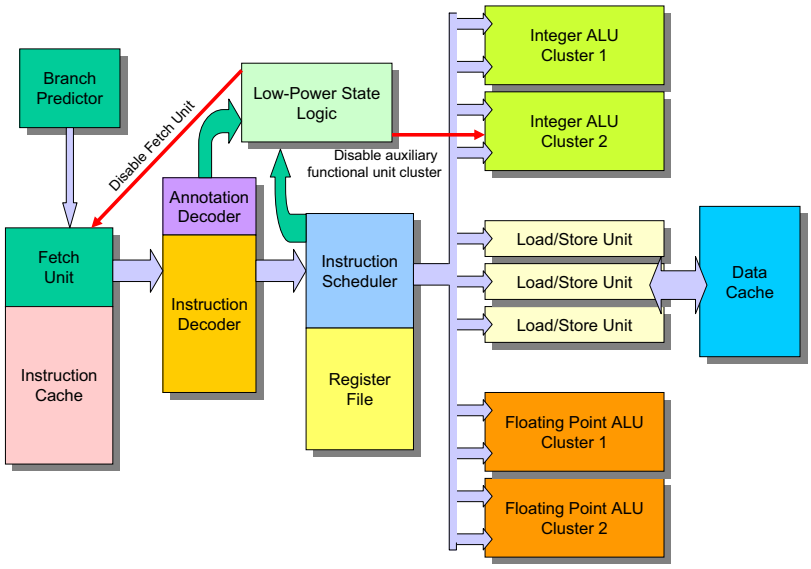


Figure 1. Pipeline organization.

3.4 Monitoring IPC and Adjusting Issue Width

The low-power state here involves reducing the issue bandwidth from 8 to 4 and disabling the second integer ALU cluster. We apply this low-power state with three different control policies based on software profiling (SW), hardware monitoring (HW), and COMB, which combines the SW and HW approaches. All of these methods make their decisions based on integer issue IPC measurements and thresholds.

The SW approach profiles application performance based on a sample run from the train dataset. It then annotates blocks of instructions that have low IPC. These annotations direct the CPU to enter or exit the low power state.

The HW method is akin to [2] and measures IPC over fixed instruction windows of 256 cycles. The processor enters the low-power state if the IPC is measured to be below a certain threshold (1.25, in our case).

We also combined these two control policies hoping to maximize their unique advantages. In this combined method, we applied the SW policy to those instruction blocks which exhibited consistently low or high performance. We utilized the dynamic HW policy for those remaining blocks that demonstrated unpredictable performance.

3.5 Monitoring Loads for Fetch Halting

Fetch halting results in power savings from the potential reduction in fetching and executing wrong path instructions and from the reduction in fetch and issue queue occupancy. With reduced occupancy rates, these processor resources may be dynamically resized more effectively to save power. In this work we concentrate specifically on identifying load instructions that miss to main memory as trigger points for fetch halting.

Fetch halting requires a combination of software profiling and hardware monitoring in order to predict which loads will miss to main memory (i.e. miss in the L2 cache), as well as to predict the criticality of a particular load. We define critical load as one that would prevent new instructions from issuing if its data was not available in the L1 or L2 caches. Stated another way, if the load that misses to main memory is not critical, then halting the fetch unit may prevent early issue of instructions not dependent on the load, thereby leading to a performance loss.

Software profiling is used to identify load instructions that have a high likelihood of becoming critical misses to main memory. We also gather various statistics for these load instructions such as IPC and queue occupancy rates while the miss is serviced, number of instructions dependent on the load, and number of hits between each miss (i.e., the *miss stride*).

Based on this profile data, we annotate load instructions that have both a high likelihood of missing as well as a high probability of being critical. Miss strides are also annotated in these instructions to help the hardware monitoring predict when the load will actually miss. These software hints are essential in aiding the load-miss predictor detect miss patterns early enough to act on. We have found that by the time a purely hardware-based miss predictor detected a miss pattern, this pattern will already have disappeared.

4 Experimental Methodology

The simulator we used in this study is derived from the SIMPLESCALAR tool suite [4]. We have added several modifications to SIMPLESCALAR to better model our reconfigurable processor. Specifically, we

- modeled single and multi-pipelined issue and execution clusters;
- added a prediction table for keeping track of load miss stride;
- added support hardware for decoding and interpreting reconfiguration annotation bits added to instructions.

Table 1. Processor resources

Parameter	Configuration
Inst. Window	128-entry RUU
Machine Width	8-wide fetch, issue, commit
Fetch Queue	32
FUs	2 Int clusters of 3 add + 1 mult/div
	2 FP clusters of 2 add + 1 mult/div/sqrt
	3 Load/Store units
L1 Icache	128KB 4-way; 64B line; 2 cycle
L1 Dcache	64KB 4-way; 64B line; 2 cycle
L2 Cache	512KB 16-way; 64B line; 7 cycle
Memory	64-bit wide; 100-cycle latency
Branch Pred.	2k 2lev + 2k bimodal + 2k meta
	3 cycle mispred. penalty
BTB	2K entry 4-way set assoc.
RAS	32 entry queue
ITLB	256 entry 8-way set assoc.
DTLB	256 entry 8-way set assoc.

Table 1 shows the complete configuration of the processor model. Note that the base case assumes a issue width of 8 and a unified 128-entry out-of-order issue queue. All comparisons in Section 5 are made to this case.

Table 2. Benchmark Fast-Forwarding

Benchmark	Train Input	Ref Input
gzip (source)	40M	40M
mgrid	400M	400M
vpr (routing)	46.1M	222M
gcc	500	500
mcf	1.5B	1.5B
equake	259M	1.15B
ammp	2B	2B
vortex	500K	500K

Our simulations were performed on a subset of the SPEC2000 integer and floating point benchmarks. The SPEC2000 benchmarks used in this study, as well as fast forwarding instruction counts (for both train and ref data sets) are given in Table 2 and are partially based on results from [17, 22]. Each benchmark was simulated for 100M instructions. These benchmarks were originally compiled using a re-targeted version of the GNU *gcc* compiler with full optimization. During software profiling on a sample run, various statistics were gathered using either the train or ref input set. We analyzed these statistics and, using automated rules-based tools, annotated instructions appropriately. These annotations, when decoded by the hardware, indicated specific reconfiguration trigger points for the hardware. In addition, an annotation may be interpreted as an absolute trigger, or as a hint to aid the hardware monitors in selecting an appropriate configuration.

5 Experimental Results

We now present experimental findings comparing performance and power results when using hardware and/or software profiling. All performance results are presented relative to the baseline processor without any power saving techniques applied. The percentage of execution time spent in a power-saving state is used as the metric for quantifying power savings.

5.1 Reducing Issue Width and Execution Units

We present results for issue width reduction (discussed in section with 3.4) with seven benchmarks: *gzip, mgrid, vpr, gcc, mcf, equake* and *vortex*.

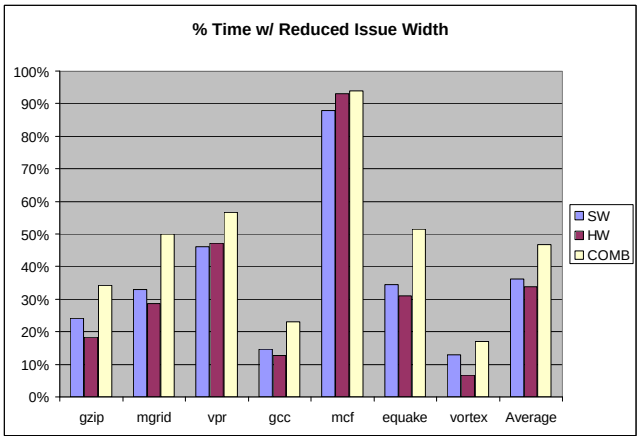
Figure 2 shows the performance impact of the software profiling, hardware monitoring, and combined software-hardware schemes The performance loss ranges from negligible to roughly 4% with an average performance loss of 2% for the individual SW and HW control policies. Combining the two approaches results in an additive effect on performance and yields an average 4% performance hit.

We can see the time spent in the low power state in Figure 3. The software and hardware methods are comparably matched and spend about 35% of the overall time in the reduced issue width mode. Combining the two approaches increases the overall time spent in the low power state to 47%.

A rough measure of energy savings is presented in Table 3—calculated as the product of percentage time in low-power state and performance. As can be seen in the table, energy savings between the two individual control policies are roughly equivalent, with the software profiling approach generally returning slightly better results. The combined SW+HW results yields significantly greater energy savings across the board at the expense of additional performance loss. This shows that the software profiling and

Figure 2. Performance impact of reduced issue width from software and hardware control policies.

Figure 3. Time spent in reduced issue width mode.



hardware monitoring approaches identify different opportunities for saving power. Our combined approach used the same threshold levels for software annotations and hardware monitoring as our individual SW and HW implementations, and this resulted in increased performance loss as well as increased power savings. Other approaches to combining SW and HW control policies may alter the threshold levels to target different performance goals.

In practice, hardware monitoring schemes generally must choose a single threshold value to apply to all applications. This single threshold works well on average, but it cannot match the performance characteristics of each program equally well. Our results show that this single threshold leads to inconsistent results among the various benchmarks. Performance-wise, the SW results are more consistent because the threshold can be tuned on a per-application basis for a target performance level. By combining the SW and HW approaches, we can select the best control policy (which we have shown to differ) that is best suited for any section of code.

Table 3. Power-performance products

Benchmark	SW	HW	COMB
gzip	23.8	18.3	33.1
mgrid	32.7	28.0	47.8
vpr	44.8	45.4	54.2
gcc	14.4	12.5	22.5
mcf	84.4	89.2	89.7
equake	34.0	30.4	49.4
vortex	12.5	6.7	16.4
Average	35.2	32.9	44.7

5.2 Fetch Halting

Unlike the fetch throttling demonstrated in [23], our approach intends to attain greater power savings by disabling the entire fetch unit for a prolonged period of time. We have modified SimpleScalar to begin fetch halting when an annotated LOAD instruction issues; the processor resumes fetching when that LOAD instruction has completed (i.e., obtained data from main memory).

5.2.1 "Perfect" Fetch Halting

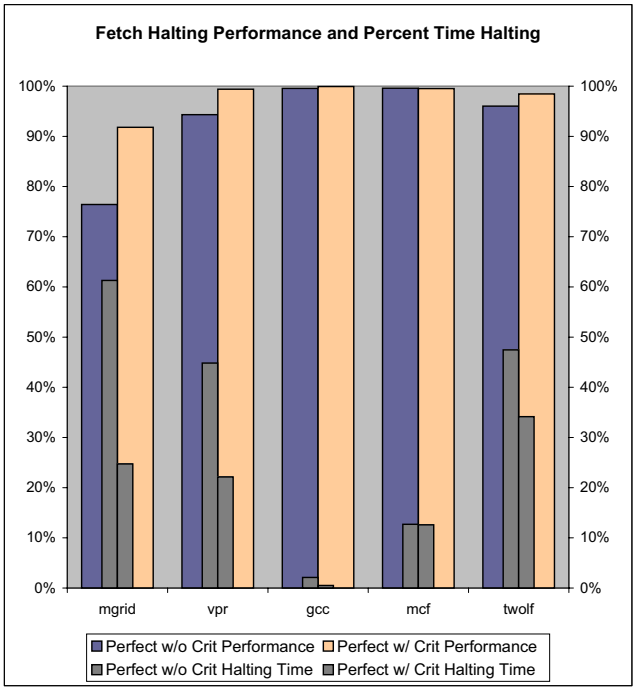
Effective fetch halting is based on two factors. The LOAD must miss to memory, and it must be on the critical path. For the purpose of simulation, we can determine whether any given instance of a load will miss, which allows us to have a "perfect" miss predictor. Table 4 shows the overall data access behavior for the test benchmarks. The level 1 and level 2 data miss rates are shown. The product of these two values yields the memory access rate for load instructions. With the exception of *gcc*, most benchmarks' memory access rates are fairly sizable. This indicates that most of these applications have the potential to significantly save power from fetch halting. On the other hand, *gcc* seldom accesses main memory, so it will not see any benefit, nor detriment, from our fetch halting approach.

Figure 4 displays our fetch halting results using a perfect load-miss predictor. The thick bars show the performance relative to the baseline processor; the thin bars show the corresponding percent time with the fetch unit disabled. For each benchmark, we show results from using a perfect load-miss predictor with and without applying a criticality predictor to restrict fetch halting. The figure shows that for some benchmarks the processor may disable the front end fetch unit for significant periods of time and have only a

Table 4. Benchmark memory access rates

	DL1 miss rate	L2 miss rate	mem access rate
mgrid	3.94%	22.75%	0.90%
vpr	4.50%	24.67%	1.11%
gcc	0.50%	12.77%	0.06%
mcf	23.78%	48.03%	11.42%
twolf	6.42%	20.08%	1.29%

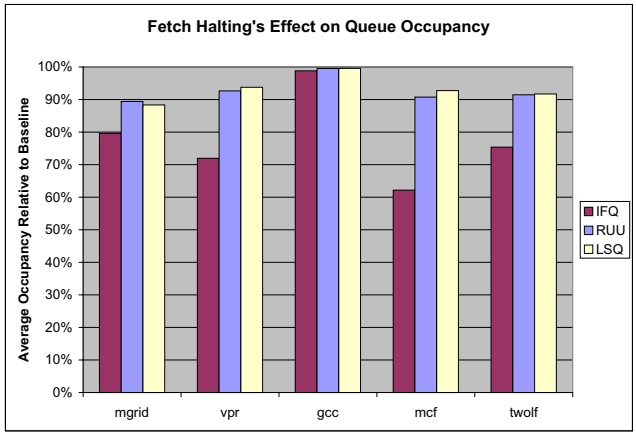
Figure 4. Fetch halting performance impact and halting time.



modest effect on performance. However, the results also show that not all load misses necessarily stall the processor. For example, *mgrid* suffers a severe 24% slowdown when halting on every miss. By restricting fetch halting to only critical loads, we are able halt the front end and save power without slowing down execution. When we incorporate our static criticality prediction with the perfect load-miss predictor, we recover most of the performance loss in *mgrid*, *vpr*, and *twolf*. However, our criticality predictor can still use some tuning as *mgrid* has a 9% performance drops even with the criticality information.

Even if fetch halting does not reduce the number of wrong path instructions fetched, disabling the fetch unit can still be advantageous in reducing the occupancy rate of various queues within the machine. For instance, with *mcf*, the average fetch and RUU occupancies dropped by 40% and 10%, respectively. Figure 5 shows fetch halting's effect on various queue occupancies in our benchmarks using the perfect miss predictor with criticalit information. By disabling the fetch unit early on when the stalling load just becomes ready to issue, we are able to maximally reduce the fetch, RUU, and load/store queue sizes and save power.

Figure 5. Fetch halting reduces occupancy rates in the fetch, issue, and load/store queues.



5.2.2 Future work

In practice, load instructions for real-world applications do not have stall behavior that can be easily predicted statically, and in the case of loads, those that do can often be prefetched. We have seen that although most load instructions exhibit some type of deterministic behavior (e.g., missing every 8th iteration), different load instructions may exhibit different cache-miss behavior. In the profiling phase each load can be categorized by behavior type and annotated appropriately. We then propose using a small hardware predictor table in conjunction with these behavior annotations to predict load stalls. The CPU can then halt the fetch unit while executing load instructions that are predicted to stall.

6 Conclusion

This paper has shown the potential for using a combined software and hardware approach to controlling processor reconfiguration for optimizing power-performance efficiency. Although software profiling and hardware monitoring both provide comparable results in our issue width reduction tests, we have shown that the two methods capture different information and may be even more effective when combined.

Our fetch halting results demonstrate the potential for large power savings in some applications. We have shown that in order for fetch halting to be effective, we must combine our criticality information gathered from software profiling with dynamic load-miss prediction.

References

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S.-T. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, and W. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceeding of the Symposium on Operating System Principles*, October 1997.

[2] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *Proceedings of the International Symposium on Computer Architecture*, July 2001.

[3] R. Balasubramonian, D. Albonesi, and A. Buyuktosunoglu. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceeding of International Symposium on Microarchitecture*, December 2000.

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

[5] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. Cook, and D. Albonesi. An adaptive issue queue for reduced power and high performance. In *Workshop on Power-Aware Computer Systems*, November 2000. Held in conjunction with the *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[6] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

[7] B. Fields, R. Bodik, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the International Symposium on Computer Architecture*, May 2002.

[8] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *28th International Symposium on Computer Architecture*, July 2001.

[9] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Workshop on Complexity-Effective Design*, June 2000. Held in conjunction with the *International Symposium on Computer Architecture*.

[10] A. Hasegawa *et al.* Sh3: High code density, low power. *IEEE Micro*, pages 11–19, December 1995.

[11] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction for high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[12] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the International Conference on Computer-Aided Design*, pages 379–386, November 2002.

[13] J. Kin, M. Gupta, and W. Mangione-Smith. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, December 1997.

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

[15] R. Maro, Y. Bai, and R. I. Bahar. Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors. In *Workshop on Power-Aware Computer Systems*, November 2000. Held in conjunction with the *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

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

[17] S. Sair and M. Charney. Memory behavior of the spec2000 benchmark suite. Technical report, IBM T. J. Watson Research Center, October 2000.

[18] R. Sasanka, C. J. Hughes, and S. V. Adve. Joint local and global hardware adaptations for energy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[19] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *International Symposium on Microarchitecture*, November 2002.

[20] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *International Symposium on High-Performance Computer Architecture*, February 2002.

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

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

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