Reducing the Energy of Speculative Instruction Schedulers

Yongxiang Liu† Gokhan Memik‡ Glenn Reinman†

†Computer Science Department, University of California, Los Angeles
‡Department of Electrical and Computer Engineering, Northwestern University

Abstract

Energy dissipation from the issue queue and register file constitutes a large portion of the overall energy budget of an aggressive dynamically scheduled microprocessor. We propose techniques to save energy in these structures by reducing issue queue occupancy and by reducing unnecessary register file accesses that can result from speculative scheduling. Our results show a 44% reduction in issue queue occupancies and an 87% reduction in register file accesses for scheduling replays. Our data show that these savings can translate into a 52% saving in issue queue energy, a 13% savings in register file energy, and a 22% overall energy savings.

1. INTRODUCTION AND MOTIVATION

In contemporary microprocessors, the out-of-order issue queue logic and register file access are responsible for a large portion of the total energy dissipated. The issue queue employs a fully associative structure that can potentially wakeup and select new instructions to issue every cycle from any slot in the queue. As a result, the issue queues are often a major contributor to the overall power consumption of the chip, and can be a hot spot [3, 4] on the core. In [7], it is estimated that instruction issue queue logic is responsible for around 25% of the total power consumption on average. Wilcox et al. [23] showed that the issue logic could account for 46% of the total power dissipation in future out-of-order processors that support speculation.

Similarly, register files also represent a substantial portion of the energy budget in modern high-performance, wide-issue processors. It is reported that modern register files represent about 10% to 15% of processor energy [18]. The development of speculative scheduling in recent microarchitectures [9, 11] will further worsen the energy dissipation in register files.

Speculative scheduling emerged in response to the growing pipeline depth in recent microprocessor designs. The pipeline depth of dynamically scheduled processors between instruction scheduling and execution – the schedule to execute (STE) window has grown to multiple cycles. For example, the recent P4 design features a 7 stage STE window [9]. Conventional schedulers broadcast instruction completion to instructions in the issue queue, and then select candidates for execution from the pool of ready instructions [21]. The throughput of such schedulers in a deep-pipelined processor is extremely low as every dependency is exposed to the depth of the STE pipeline. Speculative schedulers [9, 11], on the hand, are designed to hide the latency of the STE pipeline by anticipating operand ready time and scheduling the instruction further ahead, even before their parent instructions have completed execution.

Speculative execution works well for instructions that have predictable latencies, but load latency is highly nondeterministic. As shown in our prior work [14], this nondeterminism comes from cache misses, loads that alias with in-flight data blocks, and memory bus contention. Load instructions can take anywhere from several cycles to several hundred cycles in current generation processors. Current generation designs [9, 11] speculatively schedule load dependents by assuming cache hits. If a load misses in the cache, the processor must prevent the load's dependents from executing and then attempt to reschedule these instructions until they are correctly scheduled.

Such misscheduled instructions will speculatively access the physical register files before their operands are ready, and will therefore consume even more energy in the register file. If the latency of load instructions can be determined prior to scheduling, then the misscheduling of their dependents can be avoided, saving register file energy.

Another source of wasted energy is in the issue queue. In contemporary designs, instructions with long waiting times (i.e. dependent on long latency operations) will remain in the issue queue while waiting for their operands. Prior research [5, 4] demonstrates that an issue queue consumes power proportionately to the number of active entries in issue queue. If the long latency instructions are known a priori, we can buffer their dependents before they enter the issue queue – effectively allowing instructions to enter the scheduling window out of order. In this way, we can reduce the energy consumption in the issue queue by reducing the issue queue occupancy.

In this paper, we propose to apply load latency prediction techniques to reduce energy wasted on register file access by misscheduled instructions and energy expended in the issue queue by reducing issue queue occupancy. Our prior work [14] demonstrated how load latency can be predicted during the instruction renaming stage. But they have only considered latency prediction as a means of improving IPC performance of a conventional wakeup-and-select issue queue. In this paper, we look into modern speculative schedulers and we propose several techniques to reduce register file accesses and issue queue occupancy in such schedulers.

Particularly, in this work, we make the following contributions:

- We investigate the energy impact of speculative scheduling on deeply pipelined processors,
- We reduce the energy consumption in the issue queue and register file by applying latency prediction and instruction sorting in a speculative scheduler,
- We examine the energy cost of the added latency pre

Proceedings of the 2005 International Conference on Computer Design (ICCD'05)
0-7695-2451-6/05 $20.00 © 2005 IEEE

diction and instruction sorting hardware.

The rest of this paper is organized as follows. In Section 2 we discuss prior work, followed by a description of our energy reduction techniques in Section 3. Section 4 describes our experimental methodology. Section 5 presents our simulation results. Concluding remarks follow in Section 6.

2. RELATED WORK

Buyuktosunoglu et al. present an adaptive issue queue design by dynamically shutting down and re-enabling blocks of the issue queue [5, 4]. By shutting down unused blocks of the issue queue, they are able to proportionately reduce the energy dissipated. Our work follows this trend to scale energy dissipation with the number of active entries in the issue queue.

In conventional issue queue design, the dependents of missed loads consume a substantial amount of power while waiting for loads completion. Gschwind et al. propose a recirculation buffer for misscheduled instructions in addition to the main issue queue [8]. Similarly, Moreshet and Bahar [17] propose to use a Main Issue Queue(MIQ) and a Replay Issue Queue(RIQ). Load dependents in main queue are speculatively scheduled assuming cache hit. They will enter replay/recirculation queue if the load misses. Power is saved by reducing the main queue size relative to a baseline issue queue. As we can see, energy is still consumed when the dependents of missed loads are misspeculated, and when they wait in the replay/recirculation queue. On the other hand, our approach anticipates load misses and then before their dependents can enter the issue queue, they are buffered in low-power FIFO structures.

Ponomarev et al. present a circuit-level low-power issue queue design [19]. In their approach, energy is saved by using comparators that dissipate energy mainly on a tag match, using 0-B encoding of operands to imply the presence of bytes with all zeros and bitline segmentation. This is orthogonal to our work, and our approach can help to reduce the issue queue energy even further.

Karkhanis et al. propose to limit the number of in-flight instruction to save energy [4, 10]. In their approach, the fetch engine dynamically throttles so that instructions are not fetched sooner than necessary. This reduces the processing of speculative instructions. However, ILP is sacrificed in this approach as the number in-flight instructions is limited. In our approach, we decouple instruction fetch and instruction issuing by introducing FIFOs buffers – available ILP is still exploited in such a design.

Folegnani and Gonzalez propose to save issue queue power by disabling the wakeup of empty issue queue entries or entries that have already been woken up previously [7]. In addition, they propose to dynamically reduce the effective issue queue size by monitoring the utilization of the issue queue. Our baseline model is a more ideal version of their approach. Our approach can help this technique work more effectively by reducing the occupancy of the issue queue, thus providing more opportunity to shut down parts of the queue.

Wilcox et al. [23] demonstrate that the issue queue logic on the 8-way issue Alpha 21464 was expected to be 23% of the overall power of the core. They also argue that the issue logic could account for 46% of the total power dissipation in future out-of-order processors supporting speculation.

Lebeck et al. [13] explore an alternative means of reducing issue queue occupancy, maintaining a secondary buffer of load dependents that have been misscheduled. However, this design does not use any form of load latency prediction, and therefore will not impact register file energy. They do not explore the energy implications of this design.

Kim and Lipasti explain in detail the problem of misschedulings due to load misses, and several misscheduling recovery mechanisms [12]. We proposed latency prediction techniques to scale a conventional wakeup-and-select issue queue [14]. The Alpha 21264 uses a global 4-bit history counter to determine whether a load hits in the cache [11]. However, it is difficult to accurately predict based on global history. Memik et al [16] propose to predict load/hit miss information during load execution time to reduce scheduling replays. In this paper, we propose to predict load access time far ahead at execution stage. In this way, we prevent the dependents of long latency loads from entering the issue queue too early, saving energy both from reduced replays and from more efficient use of the issue queue.

3. SCHEDULING TECHNIQUES

In this paper, we apply latency prediction to reduce the energy from misschedulings in a speculative scheduler and the energy of the issue queue.

3.1 Conventional Wakeup-and-Select vs Speculative Scheduling

In recent microprocessor designs, the number of pipeline stages from the stage of Scheduling To Execution (STE) has grown to accommodate the latency needed for reading the register file and performing other book-keeping duties. Conventional instruction schedulers let issued instructions wake up their dependent instructions [21]. As the pipeline stages from scheduling to actual execution grow beyond a single stage, conventional wakeup-and-select can no longer schedule and execute the dependent instructions back-to-back. Each back-to-back scheduling exposes the depth of the schedule to execute window.

Modern microprocessors address the problem by speculatively waking up and selecting dependents instructions several cycles ahead [11, 9]. In this way, instructions that have back-to-back dependencies can still be executed in consecutive cycles.

3.2 Replays in Speculative Scheduling

In current microprocessors, the instructions dependent on a load (and their eventual dependents) are scheduled with the assumption that the load will hit in the cache. This assumption usually increases the performance as most of the loads actually hit in the cache. However, performance can be adversely affected in the case of cache misses.

Consider a load operation that is scheduled to execute. If the STE pipeline latency is n, and the processor is an m-way machine, **(m x (n + time_to_identify_miss))** instructions may be scheduled before the scheduler knows whether the load will hit in the cache. Once the load instruction misses in the cache, these dependent instructions will be replayed. Misscheduled instructions represent wasted issue bandwidth (as something useful could have been issued in place of the instruction) and wasted issue energy (wasted energy in the issue logic and register file).

Prior work has suggested two replay mechanisms: flush recovery (used in the integer pipeline of the Alpha 21264 [11]) and selective recovery (used by Pentium 4 [9]). In this paper we focus on selective replay, where the processor only re-executes the instructions that depend on the missed load.

A speculative scheduler needs to know when to reschedule

instructions dependent on a load miss, but load latency is highly nondeterministic. A load may hit in different levels of the memory hierarchy. In addition, a load miss may alias an already in-flight data block. Bus contention and memory port arbitration may also impact the load latency. In current processors, the latency of load can range from a few cycles to a few hundred cycles. The current strategy [9, 11] is to replay instructions every so often until the instructions are correctly scheduled. Waiting longer before replaying a misscheduled instruction can lengthen the perceived load latency. Replaying at too fine a granularity (i.e. replaying more frequently) can increase wasted issue energy and bandwidth. We have seen that replaying at the granularity of the L2 cache latency is a reasonable compromise. Therefore, on a load miss, a misscheduled dependent instruction will continue to execute every `T` cycles, where `T` is the latency of the L2 cache hit.

3.3 Latency Prediction and Sorting

In this paper, we explore the use of latency prediction techniques to reduce the number of replays in speculative schedulers. We use the lookahead latency prediction described in our previous work [14]. With correctly predicted load latency, speculative schedulers are able to schedule the dependent instructions to execute precisely after loads complete.

We propose to reduce the number of replays by allowing only instructions that are that can be executed soon to enter the issue queue, while buffering other instructions with long waiting times before they enter the issue queue. This way, instructions with long waiting times will not be mistakenly selected for execution. In addition, instead of attempting every T cycles, the scheduler makes decisions based on the accurately predicted latencies. This way, even if the dependents of a missed load enter the issue queue too early due to imperfect buffering , most of the scheduling replays can still be avoided. The speculative scheduler is also able to get better utilization out of the existing issue queue space, potentially improving performance.

Figure 1 shows the overall architecture of the proposed technique. In the early pipeline stages, we predict how long an instruction needs to wait before it can be issued, i.e, the waiting time for its operands to be produced. The "latency prediction" structure implements the techniques described in [14]. The prediction structure captures 83% of the load misses, and 99% of the cache hits. In the renaming stage, we let the PC access a load address predictor and a latency history table (LHT). The latency history table performs a last latency prediction. If the LHT can confidently report a latency, then the predicted load latency is obtained. Otherwise, we use the predicted address to access a cache miss detection engine [15] – a small, energy-efficient structure to tell if a load address will miss in a given level of the memory hierarchy. We also access the SILO (Status of In-flight Loads) structure – a small structure to tell if a load aliases with an in-flight data block. The latency of loads are predicted based on whether a load needs to access the L1 or L2 cache, main memory, or if it aliases with an in-flight data block.

The latencies of instructions other than loads are deterministic. Each instruction saves its expected completion time in a timing table [6], which can be implemented with architectural register files, so that its dependents can obtain the waiting times by checking their parents' completion times.

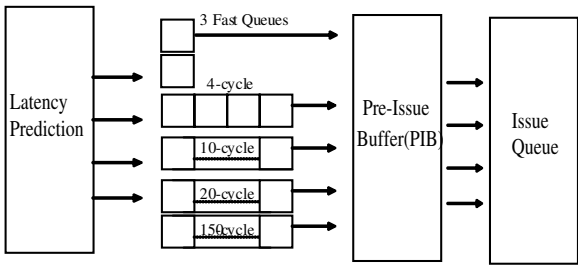

Figure 1: The Architecture to Perform Prediction, Sorting and Buffering

ILP-intensive applications	apsi,crafty,eon, gcc,gzip,vortex
Memory-intensive applications	ammp,applu,art equake,lucas,twolf

Table 1: The benchmarks used in this study.

Once its waiting time is predicted, an instruction is placed into one of the FIFO queues in the sorting engine. The primary function of the FIFO queues is to hold the instructions until their waiting time has elapsed. Instructions with very long waiting times are placed into the long queues and those with short waiting times into the short queues. Instructions should only leave the sorting queues when they can be executed soon. This way, dependents of missed loads that are correctly latency predicted will not be misscheduled. At the same time, issue queue occupancy is reduced. The sorting queues feature a locking mechanism [14] that prevents instructions from entering the issue queue before their parents.

A Preissue Buffer (PIB) is inserted between the sorting structure and issue queue to provide an inexpensive buffering of sorting instructions in cases where the issue queue fills. Instructions may *not* directly issue from the PIB - they must pass to the issue queue first.

4. METHODOLOGY

We integrate Wattch [1] with the SimpleScalar 3.0 tool set [2] to evaluate the energy performance of our design. We simulate the power and performance for a 100nm process technology at 3 GHz. We obtain the total energy dissipation and then divide it by the number of committed instructions to produce the average energy per instruction for each application.

The benchmarks in this study are taken from the SPEC 2000 suite. We rank the benchmarks by the percentage of level 2 cache misses. As listed in Table 1, we take 6 benchmarks with high miss rates as memory-intensive applications, and another 6 benchmarks with low miss rates as ILP-intensive applications. The benchmark `mcf` has extremely large miss rate and sees benefit far beyond any other benchmark from our proposed techniques. We exclude it from the memory-intensive group to make our results more representative.

The applications were compiled with full optimization on a DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital Unix V4.0. We simulate 100 Million instructions after fast-forwarding an application-specific number of instructions according to Sherwood et al. [20].

The processor configuration used for the base simulations is shown in Table 2.

Parameters	Value
Issue Width	8 instructions per cycle
ROBs	128 entries
LSQs	64 entries
Issue Queue	32 or 64 entries
Cache Block Size	L1: 32B, L2: 64B
L1 Cache	8KB, 4-way, 2-cycle latency
L2 Cache	512KB, 2-way, 12-cycle latency
Memory Latency	164 cycles
Integer FUs	8 ALU, 2 Ld/St, 2 Mult/Div
FP FUs	2 FP Add, 1 FP Mult/Div
Integer FU Latency	1/5/25 add/mult/div (pipelined)
FP FU Latency	2/10/30 add/mult/div (all but div pipelined)
Branch Predictor	4k BTB/Comb/Bimod/gshare
Branch Penalty	12, additional 2 for latency prediction

Table 2: Processor Configuration.

4.1 Structures from Proposed Techniques

We model the address predictors and latency history table (LHT) as tagless arrays as in [24, 1]. The address predictor we use has 8K entries and the LHT has 2K entries. Each entry in these structures has 40 bits. Our SILOs (Status of In-flight Loads) are modeled as a 8-entry fully associative caches, with 40 bit block sizes. We model the timing table as part of the renaming table by extending each entry of the table by 10 bits.

We model the preissue buffers and sorting queues as FIFO queues. As in Orion [22], we model FIFO queue energy with SRAM arrays [24, 1]. Our PIB has length of 64. The sorting queues have FIFO lengths of 1,5,10,20 and 150. The number of FIFOs are 3, 2, 1, 1, and 1 respectively.

Our 8-issue architecture would require 16 read ports and 8 write ports on the register file. To reduce the energy dissipation of the register file, we use a common technique: we maintain two copies of the register file, each with only 8 read ports and 8 write ports. Half of the functional units are connected to one register file and half are connected to the other. All writes go to both register files. Despite having to write each value twice (once per register file), we still save the net energy by reducing the ports on each individual register file.

5. EXPERIMENTS AND RESULTS

In this section we examine the energy data for our latency prediction engine.

5.1 Prediction, Sorting and Buffering Structures

We use Wattch to record the total energy dissipation in these structures and then divide it by the number of committed instructions to produce the average energy per instruction for each application (shown in Figure 3). Throughout this paper, we use energy dissipation per committed instruction for a fair comparison among different schedulers. The total energy from FIFOs in the sorting engine in our simulation is similar to the result obtained from Orion's FIFO implementation [22]. Overall, the energy dissipation from all additional structures for our latency prediction, sorting, and buffer engine is 0.3 nJ/instruction, which constitutes 4% of the overall energy consumption.

5.2 Speculative Scheduling

Speculative scheduling helps to hide the schedule to execute latency that is exposed with conventional wakeup and select logic. Figure 2 demonstrates that most of the benchmarks observe a large speedup with speculative scheduling, an average 60% improvement. Without speculative scheduling, back to back instructions see the full schedule to execute window. In the remainder of this section, we will focus on reducing the energy consumption of speculatively scheduled processors.

5.3 Issue Queue Energy

In the baseline speculative scheduler, instructions with long waiting times consume energy waiting for their operands in the issue queue. When we use latency prediction and instruction sorting to buffer these instructions before they enter the issue queue, we observe significant reductions in issue queue occupancy and energy. Note that this can provide benefit to any instruction that must wait for their operands in the issue queue, not just those instructions dependent on load misses.

As shown in Figure 4, we observe a 44% reduction in issue queue occupancy with a 32-entry issue queue configuration, and a even larger reduction of 53% with a 64-entry issue queue. The ILP-intensive applications observe larger reductions in issue queue occupancy because they have a larger number of in-flight instructions. The latency prediction and buffering mechanism effectively prevents these instructions from entering the issue queue earlier than necessary. In memory-intensive applications, dependents of missed loads can still enter issue queue early due to the coarser granularity of sorting queues for very long latencies.

Figure 5 shows the energy reduction in a 32-entry issue queue configuration. We observe an average of 52% reduction in issue queue energy. The ILP-intensive applications are able to reduce their issue queue energy by nearly two thirds. The memory-intensive applications also observe a large energy savings, around 45%.

5.4 Register File Energy

The baseline speculative scheduler also suffers from frequent misschedulings in the face of load nondeterminism. Without knowledge of load latencies, the scheduler has to optimistically issue load dependents to assume the load will hit in the cache. When a load misses in the cache, its dependents will have been misspeculated and will need to be rescheduled. Moreover, to avoid exposing the latency of the schedule to execute window, a speculative scheduler will replay the load's dependents at certain intervals until they are able to correctly schedule. This can impact performance in two ways: 1) by coarsening the granularity of load latencies based on the interval at which instructions replay and 2) when misscheduled instructions consume issue bandwidth that could be used to execute useful instructions. This latter component can also dramatically impact energy consumption, particularly in the issue logic and register file.

When we use latency prediction and instruction sorting to implement out-of-order entry into the scheduling window, we observe a large reduction of scheduling replays (Figure 6): the number of replays are reduced by 87% in a 32-entry issue queue configuration and by 90% in a 64-entry issue queue. The reduction comes from two sources: the dependents of missed loads that are correctly latency predicted tend not to be misscheduled because they do not enter the issue queue until they have been buffered in the sorting queues. Even

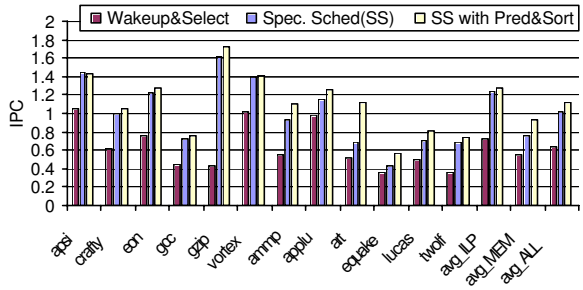

Figure 2: Performance in IPC

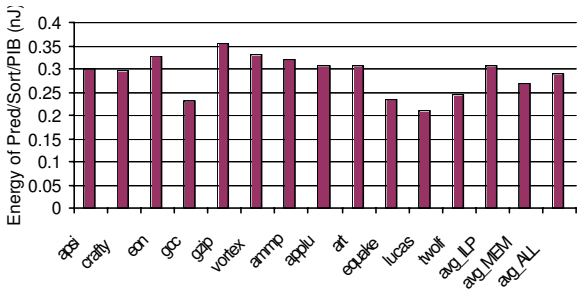

Figure 3: Energy Consumption in the Latency Prediction, Sorting and Buffering Structures (per instruction)

if the dependents of missed loads enter issue queue prematurely, misschedulings are still rare because the scheduler speculates load latency based on the accurately predicted load latencies.

The reduction in scheduling replays translates into energy savings in the register file. As shown in Figure 7, the new approach saves energy in the register file by an overall average of 13% – saving 22% in memory-intensive applications and 2% in ILP-intensive applications. Memory-intensive applications observe a much larger energy savings due to a larger amount of cache misses.

5.5 Overall Performance and Energy Reduction

When we use latency prediction and instruction sorting to implement out-of-order entry into the scheduling window, we observe a substantial improvement in IPC performance (Figure 2). On average, the memory-intensive benchmarks observe a 22% speedup. The benchmark `art` observes the largest improvement (61%). This is because these applications have frequent cache misses. With our proposed techniques, dependents of loads that miss in the cache are prevented from entering the issue queue, while other instructions with shorter waiting times are allowed to enter the issue queue earlier (and out-of-order). Therefore, the scheduler is able to better exploit memory level parallelism. The ILP-intensive benchmarks see less of an improvement because load misses are less frequent. In the case of `apsi`, we even observe a slight degradation of 1.5%. Our results show that `apsi` has relatively few cache misses. For this application, any benefit from latency prediction and sorting is canceled out by the impact of the additional pipe stages to perform latency prediction, sorting and preissue buffering.

The reduction in energy dissipation in the register file and issue logic, combined with the improved utilization of the issue queue, results in an overall drop in energy dissipation. As shown in Figure 8, the memory intensive applications observe an average 22% reduction in energy per instruction. In these applications, a large amount of misschedulings are effectively eliminated, and therefore a significant energy reduction is observed. The ILP intensive applications observe a 23% reduction. These applications, though have less energy savings from eliminated replays, have large savings in issue queue energy.

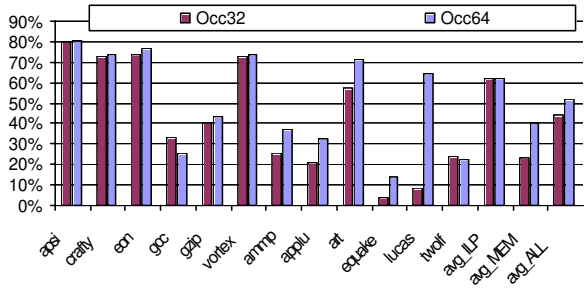

Figure 4: Reduction in Issue Queue Occupancy

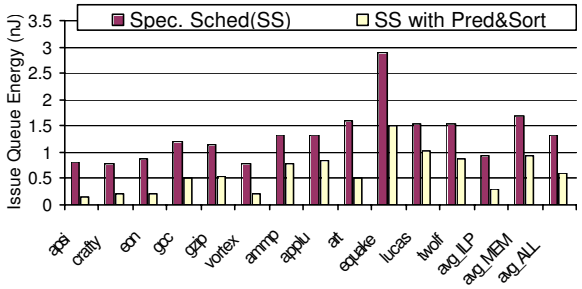

Figure 5: Issue Queue Energy Consumption Per Committed Instructions

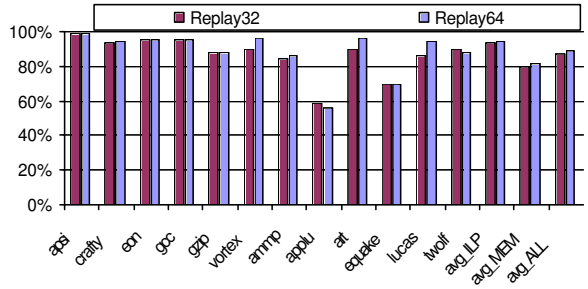

Figure 6: Reduction in Number of Scheduling Replays

6. CONCLUSION

In this paper, we use look-ahead load latency prediction and proactive instruction buffering to reduce energy consumption in issue queue and register file accesses. With predicted latencies, the schedulers can avoid unnecessarily

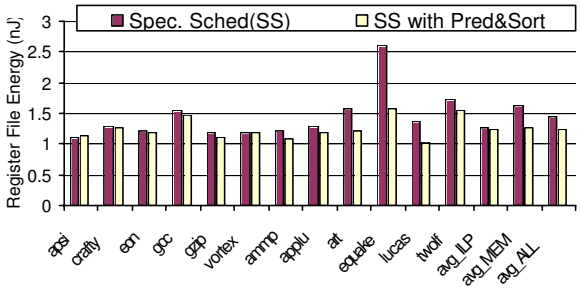

Figure 7: Register File Energy Consumption Per Committed Instructions

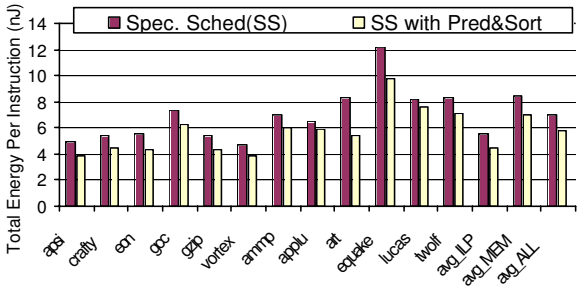

Figure 8: Total Energy Consumption Per Committed Instructions

misscheduling the dependents of missed loads. With the buffering mechanisms, we prevent instructions from entering issue queue earlier than necessary. Our results show that these savings translate into 52% savings in issue queue power, 13% savings in register file power, and 22% overall energy savings. Future work should consider the thermal implications of this reduction, as the issue logic often constitutes a processor hot spot.

7. REFERENCES

[1] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, pages 83–94. ACM Press, 2000.

[2] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[3] Alper Buyuktosunoglu, David H. Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. Power-efficient issue queue design. In *Power Aware Computing*, pages 35–58. Kluwer Academic Publishers, 2002.

[4] Alper Buyuktosunoglu, Tejas Karkhanis, David H. Albonesi, and Pradip Bose. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA'03)*, pages 147–156. ACM Press, 2003.

[5] Alper Buyuktosunoglu, Stanley Schuster, David Brooks, Pradip Bose, Peter W. Cook, and David H. Albonesi. An adaptive issue queue for reduced power at high performance. In *Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pages 25–39. Springer-Verlag, 2001.

[6] D. Ernst, A. Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, June 2003.

[7] Daniele Folegnani and Antonio Gonzalez. Energy-effective issue logic. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 230–239. ACM Press, 2001.

[8] M. Gschwind, S. Kosonocky, and E. Altman. High frequency pipeline architecture using the recirculation buffer. In *IBM Research Report(RC23113)*, 2001.

[9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[10] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED'02)*, pages 178–183. ACM Press, 2002.

[11] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[12] Ilhyun Kim and Mikko H. Lipasti. Understanding scheduling replay schemes. In *10th International Conference on High-Performance Computer Architecture (HPCA'04), 14-18 February 2004, Madrid, Spain*, pages 198–209.

[13] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, May 2002.

[14] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. Scaling the issue window with look-ahead latency prediction. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*, pages 217–226. ACM Press, 2004.

[15] G. Memik, G. Reinman, and W. H. Mangione-Smith. Just say no: Benefits of early cache miss determination. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, February 2003.

[16] G. Memik, G. Reinman, and W. H. Mangione-Smith. Precise scheduling with early cache miss detection. CARES Technical Report No. 2003_1, 2003.

[17] Tali Moreshet and R. Iris Bahar. Power-aware issue queue design for speculative instructions. In *Proceedings of the 40th Conference on Design Automation (DAC'03)*, pages 634–637. ACM Press, 2003.

[18] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture (MICRO'02)*, pages 171–182. IEEE Computer Society Press, 2002.

[19] Dmitry V. Ponomarev, Gurhan Kucuk, Oguz Ergin, Kanad Ghose, and Peter M. Kogge. Energy-efficient issue queue design. *IEEE Trans. Very Large Scale Integr. Syst.*, 11(5):789–800, 2003.

[20] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[21] R. Tomasulo. An efficient algorithm for exploring multiple arithmetic units. In *IBM Journal of Research and Development, vol. 11, no. 1, pp. 25-33, Jan. 1967.*

[22] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: a power-performance simulator for interconnection networks. In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture (MICRO'02)*, pages 294–305. IEEE Computer Society Press, 2002.

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

[24] S. Wilton and N. P. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits, pages 677-687, 1996.*