Scheduling Reusable Instructions for Power Reduction

J. S. Hu, N. Vijaykrishnan, S. Kim, M. Kandemir, and M. J. Irwin
Microsystems Design Lab
The Pennsylvania State University
University Park, PA 16802, USA
{jhu,vijay,sookim,kandemir,mji}@cse.psu.edu

Abstract

In this paper, we propose a new issue queue design that is capable of scheduling reusable instructions. Once the issue queue is reusing instructions, no instruction cache access is needed since the instructions are supplied by the issue queue itself. Furthermore, dynamic branch prediction and instruction decoding can also be avoided permitting the gating of the front-end stages of the pipeline (the stages before register renaming). Results using array-intensive codes show that up to 82% of the total execution cycles, the pipeline front-end can be gated, providing a power reduction of 72% in the instruction cache, 33% in the branch predictor, and 21% in the issue queue, respectively, at a small performance cost. Our analysis of compiler optimizations indicates that the power savings can be further improved by using optimized code.

1. Introduction

Advancing technology has increased the speed gap between on-chip caches and the datapath. Even in current technology, the access latency of the level one instruction cache can hardly be maintained within one cycle (e.g., two cycles for accessing the trace cache in the Pentium 4 [6]). In this case, pipelined instruction cache must be implemented in order to supply instructions each cycle. As a result, the pipeline depth of the front end of the datapath will increase (e.g., 6 stages in Pentium 4 [6]). Sophisticated branch predictors employed in the latest microprocessors are also very power consuming [11]. This again will increase the power contribution of the pipeline front-end.

To reduce the power consumption in the pipeline front-end, stage-skip pipeline [7][2] introduces a small decoded instruction buffer (DIB) to temporarily store decoded loop instructions that are reused to stop instruction fetching and decoding for power reduction. The DIB is controlled by a special loop-evoking instruction and requires ISA modification. Loop caches [10][1] dynamically detect loop structures and buffer loop instructions or decoded loop instructions in an additional loop cache for later reuse. A preloaded loop cache is proposed in [5] using profiling information. Loops dominating the execution time are preloaded into the loop cache during system reset based on static profiling. More generally, filter caches [9][14] use smaller level zero caches (between the level one cache and datapath) to capture tight spatial/temporal locality in cache access thus reducing the power consumption in larger level one caches.

In this paper, we propose a new issue queue design that is capable of instruction reuse. The proposed issue queue has a mechanism to dynamically detect and identify reusable instructions, particularly instructions belonging to tight loops. Once reusable instructions are detected, the issue queue switches its operation mode to buffer these instructions. In contrast to conventional issue logic, buffered instructions are not removed from the issue queue after they are issued. After the buffering is finished, the issue queue is then switched to an operation mode to reuse those buffered reusable instructions. During this mode, issued (buffered) instructions keep occupying their entries in the issue queue and are reused in later cycles. A special mechanism employed by the issue queue guarantees that the reused instructions are register-renamed in the original program order. Thus, the instructions are supplied by the issue queue itself rather than the fetch unit. There is no need to perform instruction cache access, branch prediction, or instruction decoding. Consequently, the front-end of the datapath pipeline, i.e., pipelines stages before register renaming, can be gated during this instruction reusing mode. We propose this design as a solution to effectively address the power problem in the front-end of the pipeline. Since no instruction is entering or leaving the issue queue in this mode, the power consumption in the issue queue is also reduced due to the reduced activities.

As embedded microprocessor designs are moving on to use superscalar architecture for high performance such as SandCraft MIPS64 embedded processor [12], we target at an out-of-order multi-issue superscalar processor rather than simple in-order single-issue processors that have been the focus of previous research on loop caches. Different from previous research [7][10][1][9][14], our scheme eliminates the need of an additional instruction buffer for loop caching and utilizes the existing issue queue resources. It

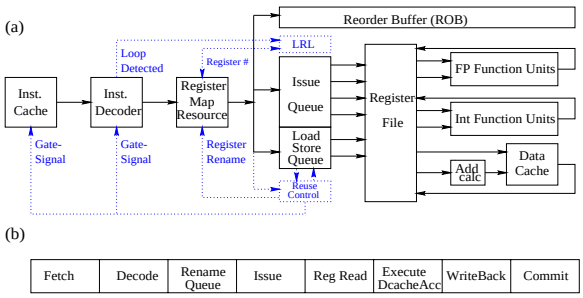

Figure 1. (a) The datapath diagram and (b) pipeline stages of the modeled baseline superscalar microprocessor. Parts in dotted lines are augmented for our new design.

automatically unrolls the loops in the issue queue to reduce the inter-loop dependences instead of buffering only one iteration of the loop in the small DIB or loop cache. Further, there is no need for ISA modification as in [7]. Note that the concept and the purpose of instruction reuse in this paper is also different from that proposed in [13]. We speculatively reuse the decoded instructions buffered in the issue queue to avoid the instruction streaming from the instruction cache rather than speculatively reusing the result of a previous instance of the instruction for performance as in [13].

The rest part of this paper is organized as follows. We present our detailed issue queue design in Section 2. Section 3 describes the experimental framework and provides the evaluation results. We investigate the impact of compiler optimizations in Section 4. Finally, Section 5 concludes the paper.

2. Modified Issue Queue Design

In this section, the detailed design of our new issue queue is elaborated. Our design is based on a superscalar architecture with a separated issue queue and re-order buffer (ROB) and the datapath model is similar to that of the MIPS R10000 [15] except that we use a unified issue queue instead of separated integer queue and floating-point queue. The baseline datapath pipeline is given in Figure 1.

The fetch unit fetches instructions from the instruction cache and performs branch prediction and next PC generation. Fetched instructions are then sent to the decoder for decoding. Decoded instructions are register-renamed and dispatched into the issue queue. At the same time, each instruction is allocated an entry in the ROB in program order. Instructions with all source operands ready are waken up and selected to issue to the appropriate available function units for execution, and removed from the issue queue. The results coming either from the function units or the data cache are written back to the register file. Instructions in ROB are committed in order.

Reusable instructions are those mainly belonging to loop structures that are repeatedly executed. Our new issue queue is thus designed to be able to reuse these instructions in the loop structures. The new issue queue design consists of the following four parts: a loop structure detector, a mechanism to buffer the reusable instructions within the issue queue, a scheduling mechanism to reuse those buffered instructions in their program order, and a recovery scheme from the reuse state to the normal state. The dotted parts in Figure 1 shows the augmented logic for this new design.

2.1. Detecting Reusable Loop Structures

To enable loop detection, we add logic to check for conditional branch instructions and direct jump instructions that may form the last instruction of a loop iteration. The loop detector performs two checks for these instructions: (1) whether it is a backward branch/jump; (2) whether the static distance from the current instruction to the target instruction is no larger than the issue queue size.

Loop detection can be performed at either the decode stage or stages after execution stage. If detection takes place at post-execution stages, the detector can be 100% sure whether it is a loop or not by comparing the computed target address and the current instruction address. However, it has several drawbacks. First, the detection may come too late for small tight loops. Second, deciding when to start buffering the detected loop can be complex. Third, the ROB has to keep the address information for each instruction in flight in order to perform this detection. On the other hand, performing loop detection at decode stage by using predicted target address has many advantages. First, loop buffering can be started immediately after a loop is detected. Second, since the instruction fetch buffer is very small (e.g., 4 or 8 entries), adding address information will not incur much hardware overhead. Further, the target address of direct jump will be available at decode stage and can be directly used for this purpose. With these tradeoffs in consideration, we choose to perform loop detection at decode stage.

2.2. Buffering Reusable Instructions

After a loop is detected and determined to be capturable (loop size less than or equal to the issue queue size) by the issue queue, we use two dedicated registers $R_{loophead}$ and $R_{looptail}$ to record the addresses of the starting and ending instructions of the loop iteration. We use a two-bit register $R_{iqstate}$ to indicate the current state of the issue queue (00-*Normal*, 01-*Loop_Buffering*, 11-*Code_Reuse*, 10-not used). A complete state transition diagram of the issue queue is given in Figure 2. The issue queue state is then changed from *Normal* to *Loop_Buffering* state. In the following cycle, the issue queue starts to buffer instructions as the second iteration begins. Our new issue queue is augmented as illustrated in Figure 3.

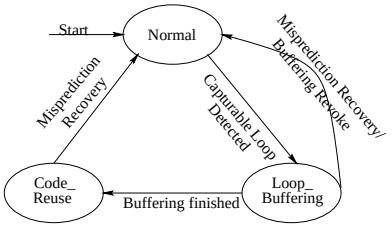

Figure 2. State machine for the issue queue.

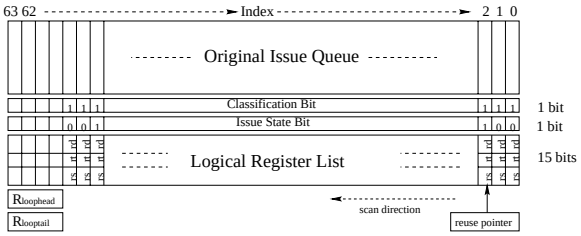

Figure 3. The new issue queue with augmented components supporting instruction reuse.

Specifically, each entry is augmented with a *classification bit* indicating whether this instruction belongs to a loop being buffered, and a *issue state bit* indicating whether a buffered instruction has been issued or not. The logical register numbers for each buffered instruction are stored in the *logical register list* (LRL). For an issue queue size of 64 entries, the additional hardware cost for these augmented components is around 136 byte (= (1 bit + 1 bit + 15 bits for three logical register numbers) * 64 / 8) cache structure.

After the issue queue enters *Loop_Buffering* state, buffering a reusable instruction requires several operations as the instruction is renamed and queued into the issue queue: the classification bit is set, the issue state bit is reset to zero, the logical register numbers of all the operands are recorded in the logical register list. With the classification bit set, the instruction will not be removed from the issue queue even after it has been issued. Note that a collapsing design is used for the issue queue.

We address two important issues concerning the buffering: when to terminate the instruction buffering and how to handle procedure calls within a loop, in the following subsections.

2.2.1. Buffering Strategy There are at least two strategies for deciding when to stop buffering and promote to *Code_Reuse* state. The first strategy is to buffer only one iteration of the loop. This scheme is simple to implement and enables more instructions to be reused from the issue queue. This is because it stops instruction fetch from the instruction cache and enters *Code_Reuse* state much earlier (at the beginning of the third iteration). In contrast, the second strategy tries to buffer multiple iterations of the loop according to available free entries in the issue queue. We use an additional counter to record the size of the current buffering iteration and to predict the size of the next iteration. After buffering one iteration of the loop, a decision is made whether the remaining issue queue can hold another iteration by comparing the counter value with the number of free entries in the issue queue. If yes, the buffering continues. Otherwise, the state of the issue queue is switched from *Loop_Buffering* to *Code_Reuse*, and the front end of the pipeline is then gated. It automatically unrolls the loop to exploit more instruction level parallelism, which is basically the way that the original issue queue works. Also, the issue queue resource is used more effectively here than in the first strategy, especially for small loops. Although the second strategy does not gate the pipeline front-end as fast as the first strategy, we choose the second one in this work for performance sake. If the execution exits the loop (check with $R_{loophead}$ and $R_{looptail}$) during the buffering state, the buffering is revoked and the issue queue switches back to the *Normal* state.

2.2.2. Handling Procedure Calls Note that the loop detector has no knowledge about either the existence or the sizes of procedure calls within a detected loop. This is because the detection only uses one iteration and happens at the end of the first iteration of the loop. If the procedure is small, the issue queue should be managed so as to capture both the loop and the procedure. Otherwise, it may not be possible to buffer the loop. Our strategy dealing with procedure calls works as follows. During the *Loop_Buffering* state, if a procedure call instruction is decoded, it will keep buffering. If the issue queue is used up before the loop-ending instruction is met, which means the procedure is too large to be captured by the issue queue, the buffering is revoked and the issue queue state is changed back to *Normal*. Otherwise, the counter value (the size of current iteration including procedure calls) is checked with the number of free entries in the issue queue to make the decision whether to promote to *Code_Reuse* state or to continue buffering.

2.3. Optimizing Loop Buffering Strategy

To avoid the state thrashing between *Loop_Buffering* and *Normal*, we introduce a small non-bufferable loop table (NBLT) holding the most recent non-bufferable loops (e.g., 8 loops). The NBLT is implemented in CAM and maintained as a FIFO queue. Each entry in NBLT has a valid bit and the address of the loop-ending instruction. If a detected loop appears in NBLT, it is identified as non-bufferable. In this case, no buffering is attempted for this loop. Otherwise, the issue queue is switched to *Loop_Buffering* state. During the *Loop_Buffering* state, if an inner loop is detected, or the execution exits the current loop, or a procedure call within the loop causes the issue queue to become full before the loop end is met, the current loop is identified as a non-bufferable loop and registered with the NBLT table. Figure 4 shows an example of an non-bufferable loop. With

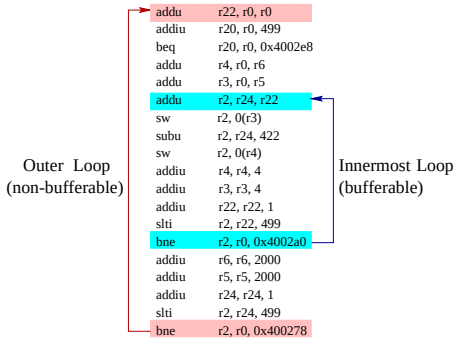

Figure 4. An example of a non-bufferable loop that is an outer loop in this code piece .

this optimization, the issue queue can eliminate most of the buffering of non-bufferable loops.

2.4. Reusing Instructions in the Issue Queue

After the reusable instructions of a loop have been successfully buffered, the state of the issue queue is switched to *Code_Reuse*. A gating signal is then sent to the fetch unit and the instruction decoder. In the following cycles, the issue queue starts to supply instructions itself by reusing the buffered instructions already in the issue queue. Thus, the instruction streaming from the instruction cache is no longer needed and the pipeline front-end is then completely gated.

During instruction scheduling, the classification bit of a ready-to-issue instruction is checked at the issue time. If this bit is not set (i.e., its value is zero meaning not a reusable instruction), the instruction is removed from the issue queue after being issued. Otherwise, the instruction still occupies its entry in the issue queue after its issue. And its corresponding issue state bit is set to indicate that this buffered instruction has been issued. The issue queue collapses each cycle if any hole is generated due to the removal of an issued instruction.

We utilize a reuse pointer to scan the buffered instructions in unidirection for instructions to be reused in the next cycle. The pointer is initiated to point to the first buffered instruction. In each cycle, the issue state bits of the first n (equal to the issue width) instructions starting from the entry pointed by the reuse pointer are checked. If the first m ($m \leq n$) bits are set, which means these m instructions have been issued and can be reused, the logical register numbers of these instructions are fetched from the logical register list and sent to renaming logic. The reuse pointer then advances by m and scans instructions for the next cycle. Renamed instructions update their corresponding entries in the issue queue. Note that only register information and ROB pointer of each instruction are updated in this case. Register renaming is needed anyway in both this scheme and conventional issue queues and hence is not an overhead. After the last buffered instruction is reused, the reused pointer is automatically reset to the position of the first buffered instruction. This process repeats until a branch misprediction is detected due to either the execution exiting the loop or the execution taking a different path within the loop. The state of the issue queue is then switched back the *Normal* state.

Note that the dynamic branch prediction is avoided during the *Code_Reuse* state. Branch instructions are statically predicted using the previous dynamic prediction outcome from *Loop_Buffering* state. The static prediction scheme works very well for loops since the branches within loops are normally highly-biased for one direction. In our scheme, the static prediction is still verified after the branch instruction completes execution. The issue queue exits *Code_Reuse* state if the static prediction is detected to be incorrect during this verification.

2.5. Restoring *Normal* State

When an ongoing buffering is revoked, if an instruction is buffered (classification bit = 1) and issued (issue state = 1), it is immediately removed from the issue queue. All classification bits are then cleared. The issue queue state is switched back to *Normal*. If a misprediction is detected at the writeback stage and the issue queue is in the *Loop_Buffering* state, a conventional recovery is carried out by removing instructions newer than this branch from the issue queue, ROB and restoring registers, followed by the recovery process of revoking the current buffering state. If a misprediction is detected in the *Code_Reuse* state, this may be due to an early branch outside the current loop, or a branch within the loop taking different path, or the execution exiting the current loop. In this case, we perform a conventional branch misprediction recovery followed by the revoking process. The gating signal is also reset when restoring the *Normal* state. It should be noted that our new issue queue has no impact on exception handling.

3. Experiments

We model the proposed issue queue using SimpleScalar 3.0 [4] and develop its power model based on Wattch [3]. The baseline configuration for the simulated processor is given in Table 1. We use a set of array-intensive applications listed in Table 2 to evaluate our new issue queue.

We find that two factors: the loop structure and the issue queue size, affect the effectiveness of our proposed issue queue design. A large loop structure cannot be completely buffered in a small issue queue. We conduct a set of experiments to evaluate the impact of issue queue size by varying it from 32 to 256 entries. In these experiments, the ROB size is set equal to the issue queue size, and the load/store queue size is half that of the issue queue. An eight-entry NBLT is used to optimize the loop detection, which helps reduce the buffering revoke rate from around 40% to 1% below.

Parameters	Configuration
Issue Queue	64 entries
Load/Store Queue	32 entries
ROB	64 entries
Fetch Queue	4 entries
Fetch/Decode Width	4 inst. per cycle
Issue/Commit Width	4 inst. per cycle
Function Units	4 IALU, 1 IMULT, 4 FPALU, 1 FPMULT
Branch Predictor	bimod, 2048 entries, RAS 8 entries BTB 512 set 4 way assoc.
L1 ICache	32KB, 2 way, 1 cycle
L1 DCache	32KB, 4 way, 1 cycle
L2 UCache	256KB, 4 way, 8 cycles
TLB	ITLB: 16 set 4 way, DTLB: 32 set 4 way 4KB page size, 30 cycle penalty
Memory	80 cycles for first chunk, 8 cycles the rest

Table 1. The baseline configuration.

Name	Source	Name	Source
adi	Livermore	tomcat	Spec95
aps	Perfect Club	tsf	Perfect Club
btrix	Spec92/NASA	vpenta	Spec92/NASA
eflux	Perfect Club	wss	Perfect Club

Table 2. Array-intensive applications.

Once the issue queue enters *Code_Reuse* state, the pipeline front-end is gated. Figure 5 shows the percentages of the total execution cycles that the front-end of the pipeline has been gated due to the instruction reuse for issue queues with different sizes. Benchmarks *aps*, *tsf*, and *wss* achieve very high gated percentage even with small issue queues due to their small loop structures. Some benchmarks work well only with large issue queues, such as *adi*, *btrix*, *eflux*, *tomcat*, and *vpenta*. An interesting observation from this figure is that increasing issue queue size does not always improve the ability to perform pipeline gating (e.g., see *tsf* and *wss*). The main reason for this case is that a larger issue queue will unroll and buffer more iterations of the loop, delaying the instruction reuse and pipeline gating. On the average, the ability to gate the pipeline front-end increases from 42% to 82% as the issue queue size increases.

Gated pipeline front-end leads to activity reduction in the instruction cache, branch predictor, and instruction decoder. Figure 6 shows the corresponding power reduction in the instruction cache ranging from 35% to 72%, branch predictor from 19% to 33%, and issue queue from 12% to 21%, as the issue queue size increases from 32 entries to 256 entries. The power reduction in the issue queue is due to the partial update (only register information and ROB pointer are updated) during the instruction reuse state in contrast to removing and inserting the instructions in a conventional issue queue. The overhead power consumption due to the logical register list, non-bufferable loop table (8 entries), and other supporting logic is also given as a percentage of the overall power consumption in Figure 6.

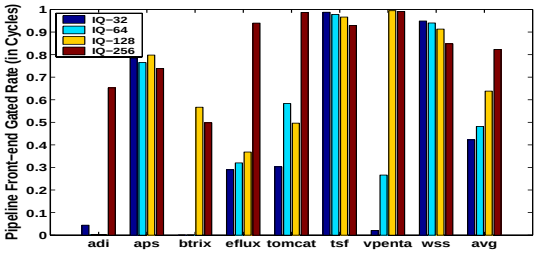

Figure 5. Percentages of the total execution cycles that the pipeline front-end has been gated with different issue queue sizes: 32, 64, 128, 256 entries.

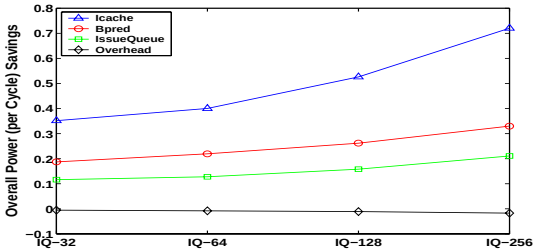

Figure 6. Power reduction in the instruction cache, branch predictor, issue queue, and overhead power consumption for different issue queue sizes.

The power reduction of the entire processor for each benchmark at different issue queue sizes is shown in Figure 7. For benchmark *adi* and *btrix*, the overall power is increased at some configurations. On the average, the power reduction is improved from 8% to 12% as the issue queue size increases. The performance impact of this new issue queue is illustrated in Figure 8. The average performance loss ranges from 0.2% (32 entry issue queue) to 4% (256 entry issue queue). This performance degradation is mainly

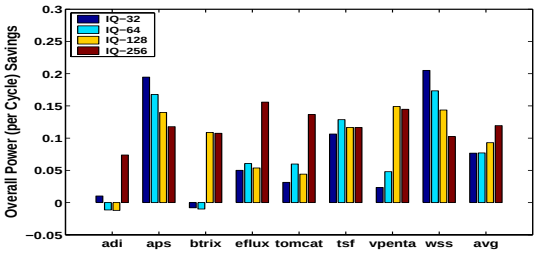

Figure 7. The overall power reduction compared to a baseline microprocessor using the conventional issue queue at different issue queue sizes.

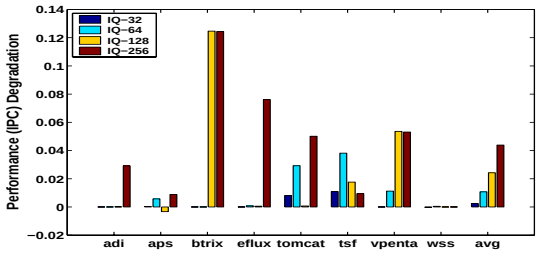

Figure 8. Performance impact of reusing instructions at different issue queue sizes.

due to the non-fully utilized issue queue (i.e., we only buffer an integer number of iterations of the loop). In benchmark *btrix*, the execution is dominated by a loop with size of 90 instructions that results in a low utilization of the issue queue with size of 128 entries or 256 entries in *Code_Reuse* state, consequently a noticeable performance loss (around 12%) as seen in Figure 8.

4. Impact of Compiler Optimizations

We notice that some benchmarks such as *adi*, *btrix*, *eflux*, *tomcat*, and *vpenta* have large loop structures, and these loops can hardly be captured with a small issue queue (e.g., with size of 32 or 64). Compiler optimizations, especially loop transformations can play an important role in optimizing these loop structures. In this work, we specifically focus on loop distribution [8] to reduce the size of the loop body.

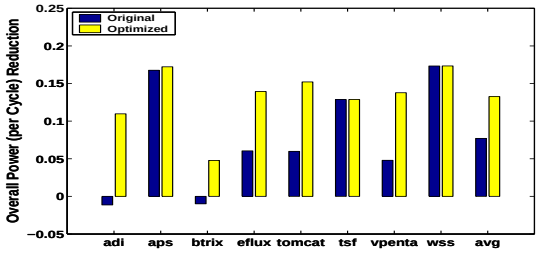

Figure 9. Impact of compiler optimizations

Figure 9 shows the comparison between optimized code (performed loop distribution) and non-optimized code, both simulated with the baseline configuration (64 entry issue queue). The average power reduction of the entire processor is increased from 8% to 13% by using the optimized code, at the cost of a slightly increased performance loss from 1% to 2%, on the average. This improvement of power reduction results from the increased percentage of gated cycles (an average from 48% to 86% (not shown due to space limit)) when executing the optimized code.

5. Conclusions

In this work, we propose a new issue queue design that is capable of buffering the dynamically detected reusable instructions, and reusing these buffered instructions in the issue queue. The front-end of the pipeline is then completely gated when the issue queue enters instruction reusing state, thus invoking no activities in the instruction cache, branch predictor, and the instruction decoder. Consequently, this leads to a significant power reduction in these components, and a considerable overall power reduction. Our evaluation also shows that compiler optimizations (loop transformations) can further gear the code towards a given issue queue size and improve these power savings.

Acknowledgments

This work was supported in part by NSF CAREER Awards 0093085 and 0093082, an NSF grant 0103583, and a grant from MARCO/GSRC-PAS.

References

[1] T. Anderson and S. Agarwala. Effective hardware-based two-way loop cache for high performance low power processors. In *IEEE Int'l Conf. on Computer Design*, 2000.

[2] R. S. Bajwa et al. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):417–424, December 1997.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proc. HPCA-6*, 2000.

[4] D. Burger, A. Kagi, and M. S. Hrishikesh. Memory hierarchy extensions to simplescalar 3.0. Technical Report TR99-25, Department of Computer Sciences, The University of Texas at Austin, 2000.

[5] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *IEEE Computer Architecture Letters*, 2002.

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technical Journal*, Q1 2001 Issue, Feb. 2001.

[7] M. Hiraki et al. Stage-skip pipeline: A low power processor architecture using a decoded instruction buffer. In *Proc. International Symposium on Low Power Electronics and Design*, 1996.

[8] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proc. the 6th ACM International Conference on Supercomputing (ICS'92*, Washington, DC, 1992.

[9] J. Kin et al. The filter cache: An energy efficient memory structure. In *Proc. International Symposium on Microarchitecture*, 1997.

[10] L. H. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proc. International Symposium on Low Power Electronics and Design*, 1999.

[11] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *Proc. the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*, February 2002.

[12] Silicon Strategies. Sandcraft mips64 embedded processor hits 800-mhz. http://www.siliconstrategies.com, 2002.

[13] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proc. the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, June 1997.

[14] W. Tang, R. Gupta, and A. Nicolau. Power savings in embedded processors through decode filter cache. In *Proc. Design and Test in Europe Conference*, 2002.

[15] K. C. Yager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.