4.1

HotSpot Cache: Joint Temporal and Spatial Locality Exploitation for I-Cache Energy Reduction

Chia-Lin Yang
Department of Computer Science and Information Engineering
National Taiwan University
Taipei 106, Taiwan
yangc@csie.ntu.edu.tw

Chien-Hao Lee
Department of Computer Science and Information Engineering
National Taiwan University
Taipei 106, Taiwan
r91017@csie.ntu.edu.tw

ABSTRACT

Power consumption is an important design issue of current embedded systems. It has been shown that the instruction cache accounts for a significant portion of the power dissipation of the whole chip. Several studies propose to add a cache (L0 cache) that is very small relative to the conventional L1 cache on chip for power optimization since a smaller cache has lower load capacitance. However, energy savings often come at the cost of performance degradation. In this paper, we propose a novel instruction cache architecture, the HotSpot cache, that achieves energy savings without sacrificing performance. The HotSpot cache identifies frequently accessed instructions dynamically and stores them in the L0 cache. Other instructions are placed only in the L1 cache. A steering mechanism is employed to direct an instruction to its allocated cache in the instruction fetch stage. The simulation results show that the HotSpot cache can achieve 52% instruction cache energy reduction on the average for a set of multimedia applications without performance degradation.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles - Cache Memories

General Terms

Performance, Design, Experimentation

Keywords

Instruction Cache, Embedded Systems, Low Power Design

1. INTRODUCTION

There has been an increasing demand for running multimedia applications on battery-operated embedded systems such as cellular phones and personal digital assistants. For a multimedia-enabled embedded system, reducing power requirement while meeting the performance demand is the most challenging design issue.

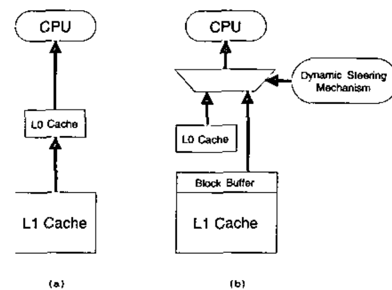

Figure 1: (a) Filter cache (b) HotSpot cache

It has been reported that the instruction cache consumes a significant portion of the total processor power. For example, 27% of processor power is dissipated in the L1 instruction cache in StrongARM 110 [13]. Cache partitioning is commonly used to reduce the dynamic energy dissipation of caches since a smaller cache has a lower load capacitance. Block buffering [14] proposes to buffer the last accessed cache line. If the data in the same cache line is accessed on the next request, only the buffer needs to be accessed. A two-phase cache access scheme can be used to avoid performance degradation [7]. The Filter cache [9] adds a bigger buffer (i.e., the L0 cache) to cache recently accessed cache blocks as shown in Figure 1(a). On each access, the L0 cache is first accessed. The L1 cache is only accessed when an L0 miss occurs. This approach can achieve more energy reduction compared with the block buffering mechanism, however, it could cause significant performance degradation if an application's working set cannot be captured in the small L0 cache. Studies show that the performance degradation could be more than 20%. In this paper, we propose a novel instruction cache architecture, the HotSpot cache as shown in Figure 1(b). Unlike the Filter cache where the L0 and L1 cache are accessed sequentially, a dynamic steering mechanism is employed to direct a request to either the L0 or the L1 cache. The L1 cache is augmented with a block buffer. The design goal is to achieve energy savings comparable to the Filter cache without sacrificing performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISLPED'04, August 9–11, 2004, Newport Beach, California, USA.
Copyright 2004 ACM 1-58113-929-2/04/0008 ...$5.00.

The energy advantage of the Filter cache comes from references that hit in the L0 cache. The L0 cache hits are results of temporal locality of frequently accessed basic blocks (i.e., hot basic blocks) and spatial locality within a cache line. Therefore, in the proposed HotSpot cache scheme, hot basic blocks are identified dynamically and stored in the L0 cache, while others are placed only in the L1 cache. The L1 cache is augmented with a block buffer to exploit the spatial locality of non-hot basic blocks for additional energy savings. To prevent performance degradation, we limit the size of basic blocks allocated to the L0 cache, and a steering mechanism is employed to direct an instruction to its allocated cache in the instruction fetch stage.

One key factor that determines the energy efficiency of the proposed HotSpot cache is the L0 cache utilization. It has been shown that program execution is often composed of distinct phases which contain different sets of hot basic blocks [12]. Multimedia applications also present the similar program behavior. Figure 2 plots frequently accessed branches running a jpeg encoder. Each data point represents a branch that is executed at least 1000 times per sample duration (10,000 branches). We can see that the jpeg encoder execution is composed of 3 phases and each phase contains different hot basic blocks. To fully utilize the L0

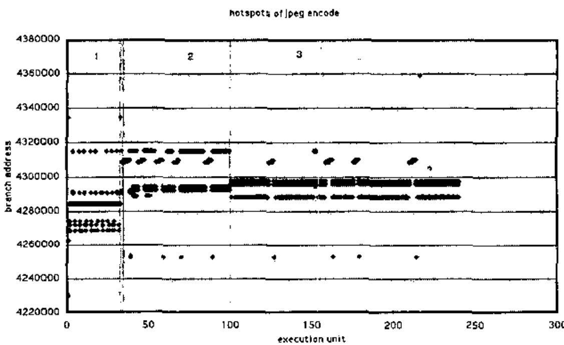

Figure 2: Frequently accessed branch distribution of jpeg encoder

cache, one should identify hot basic blocks in each program phase instead of the entire program lifetime. Bellas *et al.* [2] propose a static approach, L-Cache, that selects basic blocks to be mapped to the L0 cache based on profile information from the entire program execution. This approach may underutilize the L0 cache in program phases where identified hot basic blocks are not active. Therefore, we design a runtime mechanism that dynamically detects phase change and selects active hot basic blocks early in each program phase. To make such a hardware-based technique useful for low energy, we build the detection mechanism around the Branch Target Buffer. The simulation results show that for applications with multiple phases, the HotSpot cache can achieve up to 2x higher L0 cache utilization than the L-Cache. Without the cost of performance degradation, the HotSpot cache achieves equal or more energy savings than the Filter cache for all applications tested in the paper. The energy reduction provided by the HotSpot cache is 52% on the average.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the HotSpot cache mechanism. Section 4 describes our experimental methodology and Section 5 shows the results. Section 6 concludes this paper.

2. RELATED WORK

Several studies have proposed to use a smaller cache for reducing the energy dissipation of instruction cache[14][1][11]. The Filter cache [9] adds a smaller cache between the processor and L1 cache to store recently accessed cache blocks. As mentioned in the previous section, energy reduction is often achieved at the cost of longer average memory access time. Tang *et al.* [15] use a next address prediction scheme which dynamically predicts where the next instruction exists (L0 or L1) to reduce the performance impact of the conventional filter cache design. Bellas *et al.* [2] propose a profile-guided compiler to map frequently accessed instructions to the L0 cache. Later, they suggest to dynamically manage the L0 cache utilizing the branch predictor and the confidence estimation mechanism [16]. Basic blocks associated with high confidence branches are selected for placement in the L0 cache. Their scheme depends on the prediction accuracy of the underlying branch predictor. In this paper, we provide a more accurate approach for hot basic block selection. There are other studies proposing to dynamically detect frequently accessed instructions as well, but they have different optimization objectives from this work. Merten *et al.* [12] focus on runtime optimization, and Hu *et al.* [6] target at reducing I-cache leakage consumption.

3. HOTSPOT CACHE ARCHITECTURE

3.1 Main Idea

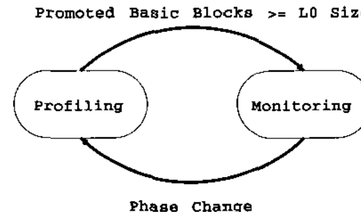

Figure 3: Hot-block selection and phase detection

The proposed system is composed of two stages as shown in Figure 3. In the profiling stage, the system gathers access frequencies of executed branches and determines which basic blocks should be promoted to the L0 cache. The promoting policy is straightforward: a basic block is promoted to the L0 cache once the corresponding branch reaches a predefined threshold (candidate threshold). To prevent performance degradation from excessive L0 cache misses, we limit the size of promoted basic blocks. Once the L0 cache is filled up, we stop profiling and enter the monitoring stage.

During the monitoring stage, the system tracks branch execution to ensure that hot branches should account for at least half of the total branches executed. If they are less than half, it indicates that either the program enters a new phase or true hot basic blocks have not yet been correctly identified. The system should go back to the profiling stage to detect new sets of hot blocks.

Merten *et al.* [12] also use a branch counting mechanism for run-time hot basic block detection. However, our scheme

has essentially different design considerations from theirs. The objective of the work done by Merten *et al.* [12] is to perform run-time optimizations on identified basic blocks. Since run-time optimizations incur time overhead, they are more conservative in declaring a hot branch. Branches with access frequencies greater than the candidate threshold are observed for a period of time before they are declared as hot branches. In contrast, we promote a basic block to the L0 cache as soon as its access frequency reaches the candidate threshold. We adopt the eager promotion policy because we want to utilize the L0 cache as often as possible. Promoting spurious hot basic blocks to the L0 cache does not incur much overhead as long as true hot basic blocks can be identified eventually. This is ensured by the monitoring scheme described above. Another design issue particular to this study is false phase change. Since we limit the size of basic blocks promoted to the L0 cache, if hot basic blocks have large static footprints, the hot branch execution percentage could be lower than 50% even though hot basic blocks have been correctly identified. If this situation occurs, the system would switch between the profiling and monitoring stages constantly. The false phase change phenomenon could potentially degrade the effectiveness of the proposed scheme if the L0 cache can not be utilized during the profiling stage. In the next section, we detail the implementations to realize the proposed scheme and our solution for the false phase change problem.

3.2 Implementations

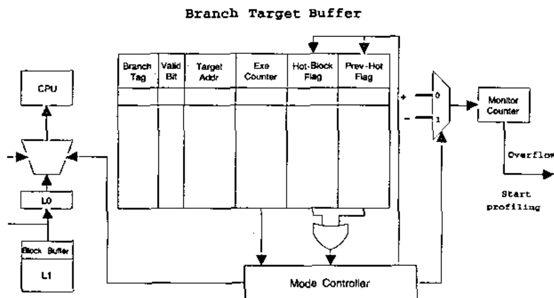

Figure 4: Block diagram of HotSpot cache

To achieve energy saving, the mechanism should not incur significant hardware overhead. Therefore, we design the hot spot and phase detection mechanisms around the Branch Target Buffer (BTB), which is commonly used in a modern microprocessor to resolve branch target addresses in the instruction fetch stage. The block diagram of the proposed scheme is illustrated in Figure 4. Each entry of the BTB is augmented with a valid bit, an execution counter, a hot-block flag, and a prev-hot flag. The valid bit indicates whether the corresponding branch is a predicted taken branch or a predicted non-taken branch. In the conventional BTB design, a branch is removed from the BTB once it is predicted non-taken. In our design, a branch is still kept in the BTB even it is predicted non-taken such that we do no lose the access frequency of the target basic block. Therefore, a valid bit is set to zero (one) for a predicted non-taken branch (predicted taken branch). The associated execution counter is updated when a branch is resolved. We only keep track of the access frequency of a taken-branch. When the execution counter reaches its maximum value (i.e., candidate threshold), a potential hot block is detected. Therefore, the hot-block flag is set and the corresponding basic block of this branch is promoted to the L0 cache.

An up/down counter called the monitor counter (8 bits) is used to track the hot branch execution percentage. The monitor counter is initially set to 128. It decrements by one when a hot branch is executed and increments by one when a non-hot branch is executed. When the counter overflows, hot-block flags are cleared and execution counters are reset. The system goes back to the profiling stage to detect new sets of hot basic blocks since identified hot branches are not active for a period of time. As mentioned before, if a program exhibits the false phase change behavior, the system would stay in the profiling stage most of time. Therefore, it is important to allow the L0 cache to be accessed during the profiling stage. We keep the hot branch information of the last phase in the prev-hot flag column until the system stabilizes (i.e., entering the monitoring stage). On each access, if either one of the hot-block and prev-hot flags is set, the access is directed to the L0 cache. Once the system enters the monitoring stage, all prev-hot flags are cleared.

The last component in the proposed mechanism is the mode controller which controls whether the L0 or L1 cache should be accessed during the instruction fetch (IF) stage. There are three fetch modes:

- L0 mode: Fetch an instruction from the L0 cache
- L1 mode: Fetch an instruction from the L1 cache
- Promoting mode: Fetch an instruction from the L1 cache and copy it to the L0 cache.

Table 1 summaries transition events for each fetch mode. If an instruction hits in the BTB and the associated valid bit is 1, the L0 mode is set if either the hot-block or prev-hot flag is 1. If both the hot-block and prev-hot flags are zeros and the execution counter equals to the candidate threshold, the Promoting mode is set and the hot-block flag is set to one. If none of the above two cases is true, the L1 mode is set. For a BTB miss or a BTB hit with the associated valid bit equal to zero, we do not change the fetch mode. The rationale is as follows. In this scenario, an instruction could be (1) a non-branch instruction, (2) a predicted non-taken branch or (3) a mis-predicted taken branch. The first case should not incur mode transition. As for the second scenario, we found that the access frequency of a non-taken branch is usually close to that of the last taken branch. Therefore, we do not keep track of the access frequency of a non-taken branch and simply let a non-taken branch inherit the fetch mode of the last taken branch. As for the third scenario, it is hard to predict the status of a mis-predicted branch (hot or non-hot), therefore, we do not change the fetch mode.

The last event causing mode transition is an L0 cache miss. Whenever a fetch misses in the L0 cache, the L1 mode is activated. This is based on the observation that if an instruction misses in the L0 cache, it is very likely that the remaining instructions in the same basic block also miss in the L0 cache. Therefore, we should fetch instructions from the L1 cache directly to avoid increasing the L1 cache access latency. Note that we switch the fetch mode to the L1 mode instead of Promoting for performance consideration. An

Fetch Mode	Events
L0 Mode	(BTB hit) && (valid bit is 1) && (hot-block flag or prev-hot flag is 1)
L1 Mode	(1) (BTB hit) && (valid bit is 1) && (hot-block flag and prev-hot flag are 0s)&& (execution counter < candidate threshold)
	(2) L0 cache misses
Promoting Mode	(BTB hit) && (valid bit is 1) && (hot-block flag and prev-hot flag are 0s)&& (execution counter == candidate threshold)

Table 1: Fetch mode transition events

Cache	Energy
L0 Cache	0.83nJ
Block Buffer	0.12nJ
L1 Cache	1.94nJ

Table 2: Cache energy consumption per access

Application	Media
ADPCM encoder/decoder	Audio
Epic/Unepic	Data
G721 encoder/decoder	Voice
Jpeg encoder/decoder	Image
Lame /Mad	Mp3
Mpeg2 encoder/decoder	Video

Table 3: Benchmark summaries

L0 cache miss indicates that the corresponding hot basic block is very likely conflicting with other hot basic blocks. Therefore, to minimize the L0 cache miss rate, once a cache line is replaced from the L0 cache, we do not bring it into the L0 cache again to eliminate conflicts among promoted hot basic blocks.

4. EXPERIMENTAL METHODOLOGY

We use the Wattch toolset [3] developed at Princeton University to conduct our experiments. Wattch generates both the performance and energy data through execution-driven simulation. We modified Wattch to simulate the HotSpot cache. Our baseline machine is a single-issue in-order processor. The processor contains a 512B, direct-mapped L0 instruction cache and a 16KB direct-mapped L1 instruction cache. The line size of both the L0 and L1 cache is 32 bytes. The BTB has 64 sets and the associativity is 4. We implemented a 2-level branch predictor with a total of 2048 entries. All the caches are single-ported. We evaluate energy consumption assuming 0.35um process technology. Table 2 shows the energy consumption per access for various instruction cache components examined in this paper. Only the dynamic energy consumption is considered in this study. Note that the HotSpot cache is orthogonal to other techniques for reducing the instruction leakage power, such as the Drowsy cache [4]. The candidate threshold value is set to 16. We performed analysis on several threshold values (8 to 1024) and found that 16 works well for all the applications tested in this paper.

Since we focus on the multimedia applications in this paper, we use applications in the Mediabench [10] and Mibench [5] to evaluate our scheme. But the proposed scheme can also be applied to other classes of applications. We choose 6 sets of encoder/decoder for different media types (data, voice, image and video). The applications tested in the study are summarized in Table 3.

5. EXPERIMENTAL RESULTS

In this section, we evaluate if the HotSpot cache successfully achieves the optimization goal: achieving energy savings comparable to the Filter cache without performance degradation. We first show that the overhead incurred from accessing various counters is negligible. We then compare the L0 cache utilization of the HotSpot cache with that of the static approach - L-cache [2]. Finally, we compare the energy and performance of the HotSpot cache with those of the Filter cache.

5.1 Overhead Analysis

The main overhead imposed by the proposed scheme is from accessing the execution counter (4 bits) associated with each BTB entry and the monitor counter (8 bits). We model a counter as a register in Wattch [3]. The energy per access is roughly 0.18pJ and 0.34pJ for 4-bit and 8-bit registers, respectively. Our simulation results indicate that for the benchmarks tested in this paper, there are 0.02/0.13 bit transition per cycle on the average for the execution/monitor counters. Note that the frequency of bit transitions of the monitor counter is larger than the execution counter because an application stays in the monitoring stage much longer than the profiling stage. On the average, the energy consumed from counter accesses is 0.048 pJ per cycle. It is roughly 5 orders of magnitude lower than the energy consumed per I-cache access (1.9nJ). Therefore, the counter overhead is negligible.

5.2 L0 Cache Utilization

One key factor that determines the effectiveness of the HotSpot cache is the L0 cache utilization. We claim that the L0 cache utilization of the HotSpot cache should be higher than that of the static approach - L-cache [2] since phase information is considered. We use the percentage of dynamic instructions accessing the L0 cache as the metric to quantify the L0 cache utilization. To obtain the L0 cache utilization for the L-cache, we sort basic blocks according to their access frequencies and add the frequencies of most frequently executed basic blocks until their accumulated code size reaches the size of the L0 Cache. This can be considered as the optimal L0 cache utilization achievable by a static mechanism similar to the L-cache since it assumes no conflicts among frequently executed basic blocks.

Two program attributes determine how well the HotSpot cache utilizes the L0 cache compared with the L-cache. The first one is whether promoted cache blocks conflict from one another in the L0 cache. Recall that replaced L0 cache lines are not brought into the L0 cache again. This is a trade-

Benchmark	No of distinct phases	L0 utilization of L-cache	L0 utilization of HotSpot cache	L0 miss rate of HotSpot cache	L0 miss rate of Filter cache
ADPCM decoder	1	99.8%	99.7%	0.0%	0.0%
ADPCM encoder	1	99.7%	99.8%	0.0%	0.0%
Unepic	11	46.3%	89.9%	0.5%	2.5%
Epic	9	77.6%	92.7%	0.5%	1.3%
G721 decoder	1	57.0%	45.1%	0.6%	11.6%
G721 encoder	1	47.2%	42.2%	1.6%	11.4%
Jpeg decoder	2	42.5%	46.1%	0.4%	11.5%
Jpeg encoder	3	43.0%	64.8%	0.5%	8.0%
Lame	3	11.3%	24.5%	0.7%	14.3%
Mad	1	73.2%	51.1%	2.6%	9.0%
Mpeg2 decoder	3	66.9%	75.2%	0.5%	4.2%
Mpeg2 encoder	7	15.1%	41.9%	0.8%	12.7%

Table 4: L0 cache utilization: HotSpot cache v.s. L-cache

off between energy savings and performance. The second factor is the number of distinct phases in a program. The proposed scheme should have a more significant advantage over the L-cache for applications with more phases. From the results shown in Table 4, we can see that for applications with more distinct phases[2], such as unepic, epic, jpeg encoder, lame, and mpeg2 decoder/encoder, the HotSpot cache achieves significantly higher L0 cache utilization than the L-cache. Adpcm encoder/decoder have very small code sizes, therefore, both the HotSpot cache and L-Cache capture the whole program in the L0 cache. Mad and g721 encoder/decoder are three applications where the HotSpot cache achieves lower L0 utilization than the L-cache. All three applications have only one phase during program execution. Mad has significantly lower L0 utilization than the L0 cache due to severe conflicts among hot basic blocks in the L0 cache.

We need to point out that the higher L0 cache utilization achieved in the HotSpot cache does not come at the expense of performance degradation. In Table 4, we list the L0 cache miss rates of both the HotSpot and Filter cache mechanism. We can see that except for mad, the L0 cache miss rates for all benchmarks are below or close to 1% while the miss rate of the Filter cache is up to 14%. Mad has severe interferences among promoted hot basic blocks. However, since we do not bring replaced hot basic blocks into the L0 cache as described in Section 3.2, conflicts among hot basic blocks do not incur excessive L0 cache misses. In the next section, we will show that performance degradation caused by L0 misses in mad is negligible.

5.3 Performance and Energy-Saving of the HotSpot Cache

In this section, we compare the energy savings and performance of the HotSpot cache with the Filter cache. Figure 5 shows the energy consumption normalized to the base configuration (without the L0 cache) for the Filter cache, block buffering, HotSpot cache, and HotSpot cache without block buffering. We can see that block buffering provides comparable energy reduction (30% to 40%) for all benchmarks. This indicates that the spatial locality within a cache line exists for all applications tested. The Filter cache achieves more energy reduction compared with block buffering for most applications, however, the advantage diminishes for applications with high L0 miss rates. For lame, the block buffering mechanism even consumes less energy than the Filter cache. The HotSpot cache is most energy efficient for all benchmarks. For applications that the Filter cache can capture almost the entire working sets (e.g., adpcm encoder/decoder and epic/unepic), the HotSpot cache successfully promotes the whole working set to the L0 cache thereby achieving equal amount of energy reduction as the Filter cache. For applications with large working sets (e.g., g721 encoder/decoder, jpeg encoder/decoder, lame and mpeg2 encoder), the HotSpot cache reduces more energy consumption than the Filter cache because of the additional energy savings from block buffering. To show the importance of exploiting the spatial locality of instructions allocated to the L1 cache, we also measure the energy consumption of the HotSpot cache without block buffering. From Figure 5, we can see that the additional energy savings provided by block buffering (HotSpot cache vs. HotSpot cache without block buffering) are significant for applications with low L0 cache utilization.

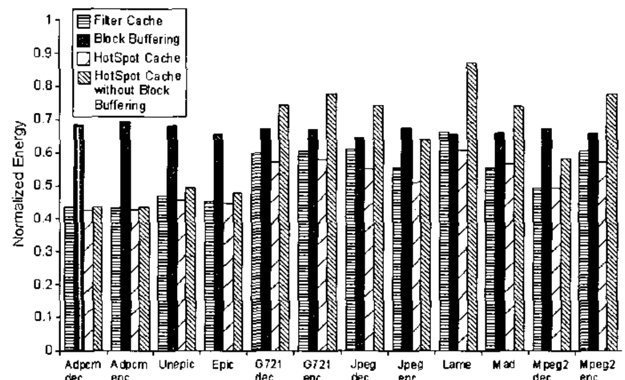

Figure 5: Normalized energy consumption of Filter cache, block buffering, HotSpot cache and HotSpot cache without block buffering

To show that the HotSpot cache reduces energy reduction without sacrificing performance, the execution time of the Filter and HotSpot cache normalized to the base con-

[2]To determine the number of distinct phases, we performed the same branch behavior analysis of jpeg encoder as shown in Figure 2.

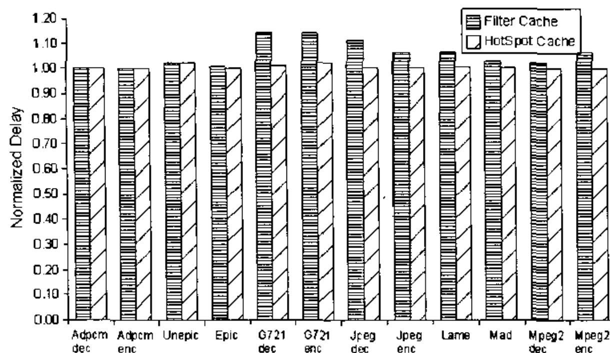

Figure 6: Normalized delay of HotSpot cache and Filter cache

figuration (without the L0 cache) is shown in Figure 6. We can see that the normalized execution time of the HotSpot cache is close to 1 for all benchmarks while the Filter cache causes up to 15% of performance degradation (e.g., g721 encoder/decoder).

6. CONCLUSIONS

In this paper, we propose an architectural approach to dynamically select basic blocks for placement in the L0 cache. We design a profiling and phase detection mechanism that can successfully identify frequently accessed basic blocks in each program phase at runtime. Only basic blocks declared as hot blocks are stored in the L0 cache. The L1 cache is augmented with a block buffer for exploiting spatial locality within a cache line for energy savings. A mode controller is employed to determine which cache (L0 or L1) should be accessed during the instruction fetch stage. The simulation results show that the proposed mechanism can achieve more energy savings than the Filter cache. The energy consumption of the instruction cache is reduced by 52% on the average for a set of multimedia applications without performance degradation.

7. ACKNOWLEDGEMENTS

This work is supported in part by research grants from ROC National Science Council (NSC-92-2213-E-002-014, NSC-93-2220-E-002-013) and Microsoft.

8. REFERENCES

[1] R. S. Bajwa, M. H. H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki, and K. Sasaki. Instruction buffering to reduce power in processors for signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4), December 1997.

[2] N. E. Bellas, I. N. Hajj, C. D. Polychronopoulos, and G. Stamoulis. Architectural and compiler techniques for energy reduction in high-performance microprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3), June 2000.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, Vancouver, British Columbia, June 2000.

[4] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: Simple techniques for reducing leakage power. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, 2002.

[5] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[6] J. S. Hu, A. Nadgir, N. Vijaykrishnan, M. J. Irwin, and M. Kandemir. Exploiting program hotspots and code sequentiality for instruction cache leakage management. In *Proc. of the International Symposium on Low Power Electronics and Design (ISLPED'03)*, Seoul, Korea, August 2003.

[7] M. B. Kamble and K. Ghose. Analytical energy dissipation models for low power caches. In *Proc. of the International Symposium on Low Power Electronics and Design*, 1997.

[8] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA)*, Goteborg, Sweden, June 2001.

[9] J. Kin, M. Gupta, and W. H. Mangione-Simith. The filter cache: An energy efficient memory structure. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, December 1997.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Media-bench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on MicroArchitecure*, December 1997.

[11] L. H. Lee, W. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of International Symposium on Low Power Design*, pages 63–68, August 1999.

[12] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of International Symposium Computer Architecture*, pages 136–147, May 1999.

[13] J. Montanaro and et al. A 160-mhz, 32-b, 0.5-w cmos risc microprocessor. *IEEE Journal of Solid State Circuits*, 31(11):1703–1714, November 1996.

[14] C.-L. Su and A. Despain. Cache design tradeoffs for power and performance optimization: A case study. In *Proceedings of International Symposium on Low Power Design*, April 1995.

[15] W. Tang, R. Gupta, and A. Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures. In *International Conference on Computer Design(ICCD)*, Austin, Texas, USA, 2001.

[16] N. E. Bellas and I. N. Hajj and C. D. Polychronopoulos. Using Dynamic Cache Management Techniques to Reduce Energy in General Purpose Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(6), December 2000.