# HotSpot Cache: Saving I-Cache Energy with Dynamic Program Hot Spot Detection for Multimedia Applications

**Chia-lin Yang, Chien-hao Lee**

Department of Computer Science and Information Engineering

National Taiwan University

Taipei 106, Taiwan

**Abstract** Power consumption is an important design issue of current embedded systems. It has been shown that instruction cache accounts for a significant portion of the power dissipation of the whole chip. Several studies have proposed 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 mechanism that detects program hot spots dynamically and stores only hot spots in the L0 cache. The optimization goal is to achieve high L0 cache utilization without sacrificing performance. We design a run-time hot-spot detection mechanism around the Branch Target Buffer. The results show up to 57 % energy reduction without performance degradation for a set of multimedia applications.

## 1 Introduction

There has been an increasing demand for running multimedia applications on battery-operated embedded system such as cellular phones and personal digital assistants. The most challenging design issue for such systems is how to reduce energy consumption while meeting the performance demand of multimedia applications.

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 [1]. Cache partition is commonly used to reduce cache energy dissipation since a smaller cache has a lower load capacitance. The Filter cache [2] proposes to add a relative small cache (L0 cache) between the CPU and L1 cache as shown in Figure 1.1. On each access, the L0 cache is first accessed. The L1 cache is only accessed when an L0 miss occurs. Because the L0 cache size is very small, the miss rate is high and the performance degradation can be more than 20%.

One approach to amend this performance degradation is to store only frequently

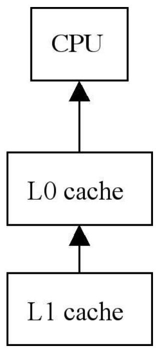

Figure 1.1: Filter Cache

accessed instructions in the L0 cache and bypass the L0 cache for other instructions. L-Cache, a compiler-managed L0 cache [8], maps frequently accessed basic blocks to the L0 cache based on the profile information from the program's entire execution. The downside of a static approach is that it is not able to adapt to changes in program behavior. It has been observed that program execution often occurs in distinct phases, which may contain different sets of hot basic blocks [9]. Therefore, to utilize the L0 cache more efficiently, we should identify hot basic blocks in each distinct phase instead of the entire program execution.

In this paper, we propose a dynamic mechanism that is able to detect program phase change and select hot basic blocks early in each program phase. The proposed approach contains two stages: profiling stage for hot basic block detection and monitoring stage for phase change detection. In the profiling stage, execution frequencies of branches are tracked and basic blocks corresponding to frequently accessed branches are promoted to the L0 cache. Once the size of promoted basic blocks exceeds the L0 cache size, the system enters the monitoring stage for phase change detection. We limit the number of instructions promoted to the L0 cache for performance consideration. Since performance is important for multimedia applications, our design goal is to achieve high L0 cache utilization without sacrificing performance. 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 the proposed scheme reduces the instruction cache energy consumption by 50% on the average without sacrificing performance.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 provides the motivation behind our approach. Section 4 details the proposed mechanism. Section 5 describes our experimental methodology and Section 6 presents the results. Section 7 concludes this paper.

## 2 Related work

Several studies have proposed to use a smaller cache for reducing the energy dissipation of instruction cache. Block buffering [3] uses a block buffer to latch the last cache line. [6] suggests to buffer decoded instruction to save energy both in the I-cache access and instruction fetch/issue logic. Lee *et al.* [10] introduces a loop cache that dynamically fills the cache after detecting a simple loop.

The filter cache [2] adds a smaller cache between the processor and L1 cache to store recently accessed blocks. As mentioned in the previous section, energy reduction is often achieved at the cost of longer average memory access time. Bellas *et al.* [9] use a profile-guided compiler to map frequently accessed instructions to the L0 cache. Later, Bellas *et al.* [7] propose to identify hot spots dynamically. Their scheme is based on the observation that highly predictable branches (high confidence branch) tend to be accessed more frequently than others. Therefore, basic blocks associated with high confidence branch are selected for storing in the L0 cache. Their scheme depends on the prediction accuracy of underlying branch predictor. Therefore, it may fail to identify frequently executed basic blocks if the branch behavior of an application is not predictable. For examples, the ADPCM encoder/decoder have very small code sizes, however, the branch mis-prediction rate is up to has up to x%. In this paper, we provide a more accurate approach for hot basic block selection and limit the size of instructions stored in the L0 cache to avoid performance degradation. Weiyu Tang *et al.* [15] proposed 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. The effectiveness of their scheme depends on the prediction accuracy. Later in this paper, we will compare the proposed hotspot cache with their mechanism. Similar to this study, Merten *et al.* [9] also propose to dynamically identify hot basic blocks in each program phase. But their objective is to perform runtime optimization. Also J. S. Hu *et al.* [15] propose to dynamically identify hot basic blocks using a branch target buffer(BTB), but their objective is to provide a leakage management approach.

## 3 Motivation & Approach

In this section, we first analyze the branch execution behavior of multimedia applications that motivate this study. After that, we describe the main idea of proposed approach to reduce the instruction cache energy consumption without causing performance degradation.

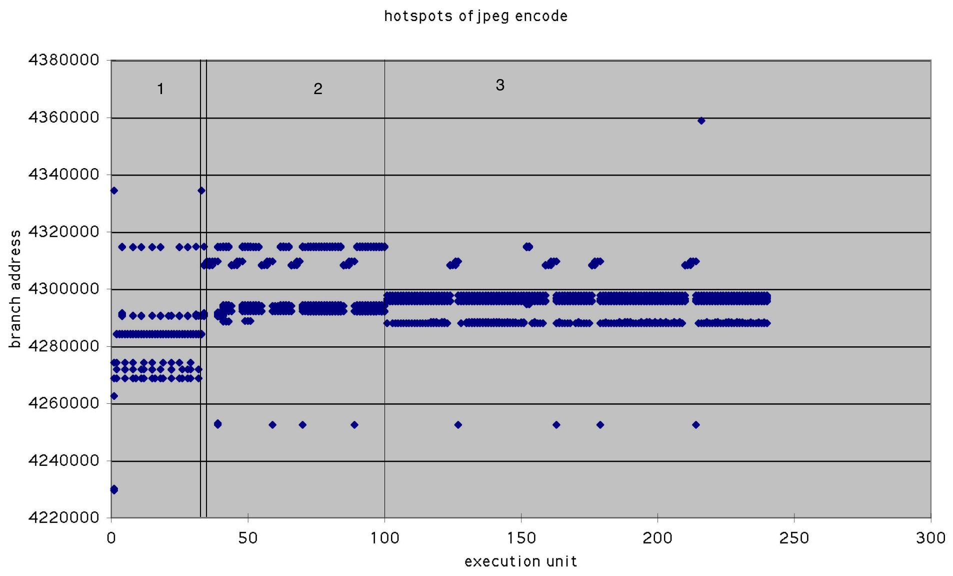

Figure 3.1: Frequently accessed branch distribution of jpeg encoder

### 3.1 Program Behavior

It has been shown that program execution often occurs in distinct phases, which may contain different set of hot basic blocks [9]. Multimedia applications also present the similar program behavior. Figure 3.1 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.

Bellas *et al.* [9] proposes a static approach that selects basic blocks to be mapped to the L0 cache based on the profile information from the entire program execution. To avoid performance degradation, the number of basic blocks mapped to the L0 cache is limited by the capacity of the L0 cache size. This approach may underutilize the L0 cache in program phases where identified hot basic blocks are not active. To fully utilize the L0 cache, one should identify hot basic blocks in each program phase instead of the entire program lifetime. In this paper, we propose a run-time mechanism that dynam-

Figure 3.2: Run-Time L0 Cache Management Scheme

The proposed system is composed of two stages as shown in Figure 3.2. 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 quite straightforward: a basic block is promoted to the L0 cache once the corresponding branch reaches a predefined threshold (candidate threshold). We assume that most frequently accessed basic blocks are more likely to reach the candidate threshold earlier than others. To prevent performance degradation from excessive L0 cache misses, we limit the size of promoted cache blocks. Once the L0 cache is filled up, we stop profiling and enter the monitoring stage.

Note that our method is quite aggressive in declaring a hot branch. In [9], which dynamically identifies hot blocks for runtime optimization, branches with access frequency greater than the candidate threshold (candidate branches) are observed for a period of time. Candidate branches are declared as hot branches when they are active during that period and the candidate branches account for at least a certain percentage of the total branches executed during that time. In contrast, we promote a basic block to the L0 cache as soon as its access frequency reaches the candidate threshold. The main reason for choosing the eager promotion policy is because we would like to utilize

Figure 4.1: Block Diagram of Proposed Run-Time L0 Cache Management Scheme

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 modern microprocessor to resolve branch target address in the instruction fetch stage. The block diagram of proposed scheme is illustrated in Figure 4.1. Each entry of BTB is associated with an execution counter, a hot-block flag, and a prev-hot flag (which will be explained later). For each BTB hit, the associated execution counter is incremented. 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.

The phase detection mechanism is similar to what proposed in [9]. 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 counts down when a hot branch is executed and counts up when a non-hot branch is executed. When the counter saturates, it means that non-hot branches account for more than half of executing branches. This indicates that either the program enters a new phase or we have not correctly identified hot blocks. The system should enter profiling stage at this point to detect new sets of hot blocks. The hot-branch flags to set in BTB are switch to the other one (either hot-block flag to prev-hot flag, or prev-hot flag to hot-block flag) and the profiling unit is turned on again.

A potential problem of the proposed phase detection mechanism is false phase change. Since we limit the size of cache blocks promoted to the L0 cache to avoid performance degradation, if hot basic blocks have large static footprints, the hot branch execution percentage could be lower than 50% even though identified hot blocks are correct. If this situation were to occur, the system would switch between profiling and monitoring phase constantly. The false phase change phenomenon could potentially degrade the effectiveness of the proposed scheme since there is a warm-up period at each new profiling stage when the L0 cache is not utilized. To solve this problem, we keep the hot branch information of the previous phase until the system stabilizes (i.e., entering the monitoring stage). The hot-spot information is kept in the prev-hot flag. 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 should be cleared. To avoid the overhead of copying the hot-flag fields to the prev-hot flag for each new phase, these two fields are alternately selected as the destination for setting hot-branch.

The last component in the proposed mechanism is the mode controller which controls whether the L0 or L1 cache should be accessed in 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

| Fetch Mode | Events |
|---|---|
| L0 Mode | (1) BTB hit && (in profiling phase) && (hot-spot flag or prev-hot flag) is 1 |
|  | (2) BTB hit && (in monitoring phase) && hot-spot flag is 1 |
| L1 Mode | (1) BTB hit && (in monitoring phase) && hot-block flag is 0 |
|  | (2) BTB hit && (in profiling phase) && (hot-block flag and prev-hot flag are 0) && (execution counter ++ < candidate threshold) |
|  | (3) L0 cache misses |
|  | (4) Branch mis-prediction |
| Promoting Mode | BTB hit && (in profiling phase) && (hot-block flag is 0) && (execution counter ++ == candidate threshold) |

Table 4.1: Fetch Mode Transition Events

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

Table 4.1 summaries transition events for each fetch mode. Below we elaborate on these events. On every BTB hit,

1. If the hot-block flag is 0 and in profiling stage, if prev-hot flag is 0, the corresponding basic block has not been promoted to the L0 cache, neither in previous phase. Therefore, an instruction should be fetched from the L1 cache. The remaining question is whether to copy the instruction to the L0 cache (i.e., L1 mode or promoting mode). The execution counter is incremented by one. If an overflow occurs (i.e. the execution counter value is equal to the candidate threshold), the corresponding basic block should be promoted to the L0 cache (promoting mode); otherwise, the L1 mode is set; else if prev-hot flag is 1, the corresponding basic block is probably still in L0 cache. So, L0 mode is set.

2. If the hot-block flag is 0 and in monitoring stage, the corresponding basic block has not been promoted to the L0 cache. Therefore, an instruction should be fetched from the L1 cache. The L1 mode is set.

3. If the hot-block flag is 1, the corresponding basic block has been promoted to the L0 cache. Therefore, an instruction should be fetched from the L0 cache (L0 mode).

If an instruction misses in the BTB, it could be (1) a non-branch instruction, (2) a non-taken branch or (3) a taken branch. The first two cases should not incur mode transition since the access frequency of the executing instruction should be the same

| Application | Type |
|---|---|
| ADPCM encoder/decoder | Audio compression/decompression |
| Epic/Unepic | Data compression/decompression |
| G721 encoder/decoder | Voice compression/decompression |
| Jpeg encoder/decoder | Image compression /decompression |
| Lame /Mad | Mp3 compression /decompression |
| Mpeg2 encoder/decoder | Video compression /decompression |

Table 5.1: Benchmark Summaries

as the previous taken-branch, which determines the current fetch mode. The third scenario could happen either because the branch is replaced from the BTB or the branch prediction is not accurate. To prevent a hot branch being replaced from the BTB, we modify the BTB replacement policy such that a non-hot branch should be first replaced. Therefore, a miss taken-branch is very likely to be a non-hot branch. If a branch is mis-predicted, it is very likely that it is not executed frequently therefore the L1 mode should be activated. In either case, mis-prediction is detected once the branch is resolved. Therefore, the fetch mode should transition to the L1 mode if a mis-prediction occurs.

Finally, 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 shall fetch instructions from the L1 cache directly to avoid increasing the L1 cache access latency. Note that once a cache line is replaced from the L0 cache, we do not bring it into the L0 again to eliminate conflicts among promoted hot basic blocks for performance consideration.

Note that our instruction cache is enhanced with one entry line-buffer as proposed in [14]. A requested cache block is fetched into the line buffer. Accessing the line buffer occurs in parallel with the decoding of the L1 cache to avoid performance degradation. If a memory reference hits in the line buffer, data are read from the L1 cache to achieve energy savings. Since the instruction stream usually presents good spatial locality, the line buffer is very effective to further reduce the energy consumed from those non-hot basic blocks.

## 5 Experimental Methodology

We use Wattch toolset [5] 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 energy-saving techniques proposed in this paper and also incorporated newer cacti library. Our baseline machine model is an ARM-like 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 select the cache and BTB configuration based on the SA-1110 design [4]. Since DRAM memory power is not modeled in the Wattch toolset, a four-way 512KB L2 cache is used as a backing storage. All the caches are single-ported. We evaluate energy consumption assuming 0.35um process technology and activity sensitive conditional clocking. The candidate threshold value is set to 64. We perform analysis on several threshold values (8 to 1024) and find 64 works well for all the applications.

Since we focus on the multimedia applications in this paper, we use applications in the Mediabench [11] and Mibench [12] to evaluate our scheme. But our 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 5.1.

## 6 Experimental Results

In this section, we evaluate if the proposed hot-spot I-cache successfully achieves the optimization goal: achieving high L0 cache utilization with performance guarantee.

### 6.1 Overhead Analysis

Before presenting the performance impact and energy savings of the proposed hot-spot I-cache, we first analyze the energy overhead from accessing the execution counter (5 bit) associated with each BTB entry and the monitor counter (8 bits). We model a counter as a register in Wattch [5][*] . The energy per access is roughly 0.21pJ and 0.34pJ for 5-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 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 0.046 pJ per cycle. It is roughly 7 orders of magnitude lower than the energy consumed per I-cache access (0.19mJ). Therefore, the

[*]Kaxiras *et. al.* [13] use the same method to evaluate the counter overhead incurred by their scheme.

| Benchmark | Number of distinct phases | % dynamic instructions of a perfect static mechanism | % dynamic instructions of hot-spot I-Cache | L0 cache miss rate of hot-spot I-Cache | L0 cache miss rate of the filter cache |
|---|---|---|---|---|---|
| ADPCM decoder | 1 | 99.8% | 99.7% | 0.0% | 0.0% |
| ADPCM encoder | 1 | 99.7% | 99.7% | 0.0% | 0.0% |
| Unepic | 11 | 46.3% | 88.3% | 0.5% | 2.5% |
| Epic | 9 | 77.6% | 94.6% | 0.3% | 1.3% |
| G721 decoder | 1 | 57.0% | 49.2% | 1.1% | 11.6% |
| G721 encoder | 1 | 47.2% | 43.3% | 1.5% | 11.4% |
| Jpeg decoder | 2 | 42.5% | 46.0% | 0.4% | 11.5% |
| Jpeg encoder | 3 | 43.0% | 65.0% | 0.4% | 8.0% |
| Lame | 3 | 11.3% | 20.1% | 0.5% | 14.3% |
| Mad | 1 | 73.2% | 50.7% | 2.5% | 9.0% |
| Mpeg2 decoder | 3 | 66.9% | 74.7% | 0.3% | 4.2% |
| Mpeg2 encoder | 7 | 15.1% | 42.5% | 1.1% | 12.7% |

Table 6.1: Summaries of important program attributes

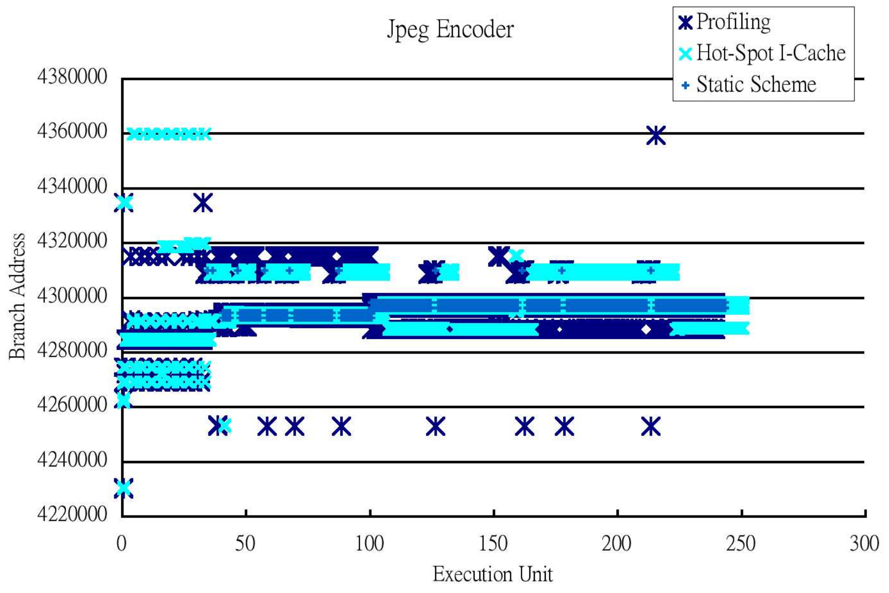

Figure 6.1: Hot branches in HotSpot cache v.s. profiling-based frequently executed Branches v.s. static scheme. Note that the HotSpot Cache stays in the monitoring phase during the execution interval where no data points of HotSpot Cache are plotted.

counter overhead is negligible.

## 6.2 Miss rates and Access Frequency of the L0 cache of the HotSpot I-Cache

Before presenting the energy savings and performance impact of the HotSpot Cache, we first evaluate its effectiveness using the L0 cache miss rate (# of L0 cache misses / total # of memory references) and the percentage of dynamic instructions accessing the L0 cache (i.e., L0 cache access frequency). These two metrics can best quantify the effectiveness of the proposed scheme independent of the underlying technology. Recall that our optimization goal is to achieve high L0 cache utilization without sacrificing performance. It is simple to demonstrate the performance guarantee since a low L0 cache miss rate implies negligible performance degradation incurred. The difficult part is to evaluate how well the proposed scheme utilizes the L0 cache provided that it does not incur significant performance degradation. To find a comparison base, we sort basic blocks according to their access frequency and add the frequencies of most frequently executed basic blocks provided that their sizes is smaller than the size of the L0 cache. This can be considered as the optimal L0 cache utilization achievable by a static mechanism since it assumes no conflicts among frequently executed basic blocks. Table 6.1 summaries the simulation results for all 12 benchmarks tested in this paper.

We first examine whether the proposed scheme meets our performance criteria. We list the L0 cache miss rates of the filter cache mechanism in Table 6.1 for comparison. 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%. Since we limit the size of promoted cache blocks, majority of the L0 misses come from conflicts among promoted cache blocks. As mentioned in Section 4, to reduce conflict misses, once a cache line is replaced from the L0 cache, we do not bring it back into the L0 cache again. This is very important in minimizing performance degradation. Note that we are not able to further reduce the L0 cache miss rate since the branch associated with the replaced cache line is still marked as a hot branch. This implies that the first instruction of a cache block that misses in the L0 cache will still result in a L0 cache miss when it is executed again. We think 2.5% of L0 cache miss rate should not incur significant performance degradation. We will show the performance impact of the proposed scheme in terms of execution time in the section below.

Two program attributes determine how well the proposed scheme can utilize the L0 cache compared to perfect static mechanism described above. 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 design decision is to trade-off energy savings with performance. The second factor is the number of distinct phases in a program. The proposed scheme should have more significant advantage over a static approach for applications experiencing more phase changes (See Figure 6.1).

We now compare the L0 cache access frequency of the proposed scheme with that of a perfect static mechanism listed in Table 6.1. Mad, g721 encoder/decoder are three applications with lower L0 utilization than a static one because of conflicts among frequently executed basic blocks in the L0 cache and they have only one phase. For Mad, we examine the addresses of 16 most frequently executed basic blocks identified by the static approach and find 12 of them conflict with each other. For applications with more distinct phases[†] , such as unepic, epic, jpeg encoder, mpeg2 decoder, and mpeg2 encoder, the HotSpot Cache mechanism achieves significantly higher L0 cache utilization than a perfect static mechanism (between 8% to 42% differences). For applications that have only one phase during execution, such as adpcm encoder/decoder, the L0 cache utilization of the HotSpot Cache is close to a perfect static mechanism as expected. Note that adpcm encoder/decoder have very small code sizes, therefore, both static and hot-spot cache capture the whole program in the L0 cache.

To illustrate the success of the proposed scheme, we plot the branches promoted to the L0 cache in both the static scheme and hot-spot cache in Figure6.1 for the jpeg encoder. We also show frequently executed branches (i.e., a branch that is executed at least 1000 times per sample duration)identified through profiling in each phase for verification. Each data point associated with the HotSpot Cache represents an identified hot branch. We can observe that majority of identified hot branches of the hot-spot cache overlap with frequently executed basic blocks through profiling. On the other hand, the static scheme only identifies parts of these branches in two out of three phases. This explains why the hot-spot cache can achieve 22% more dynamic accesses compared to the static scheme.

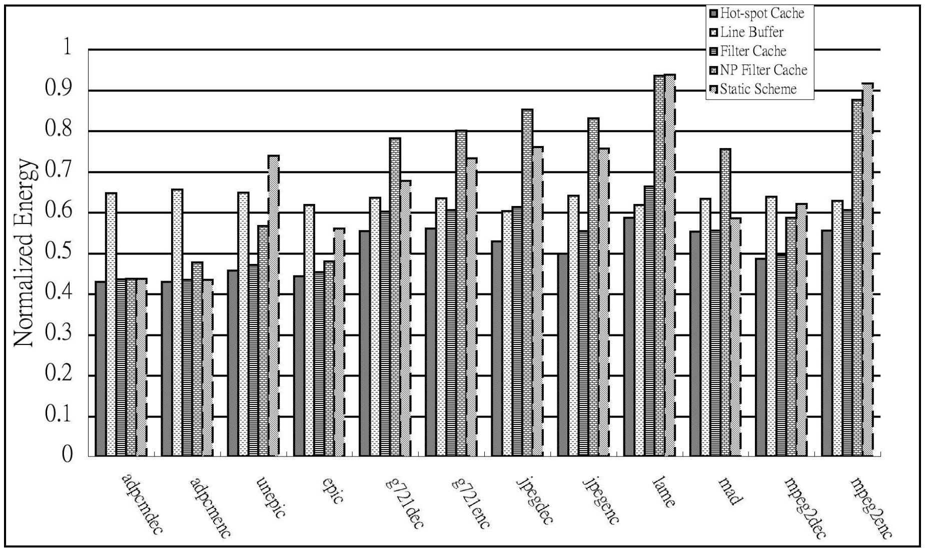

Figure 6.2: Normalized Energy Consumption of HotSpot Cache, L1 cache with 1 entry line buffer, Filter Cache, NP Filter Cache, and Static Scheme

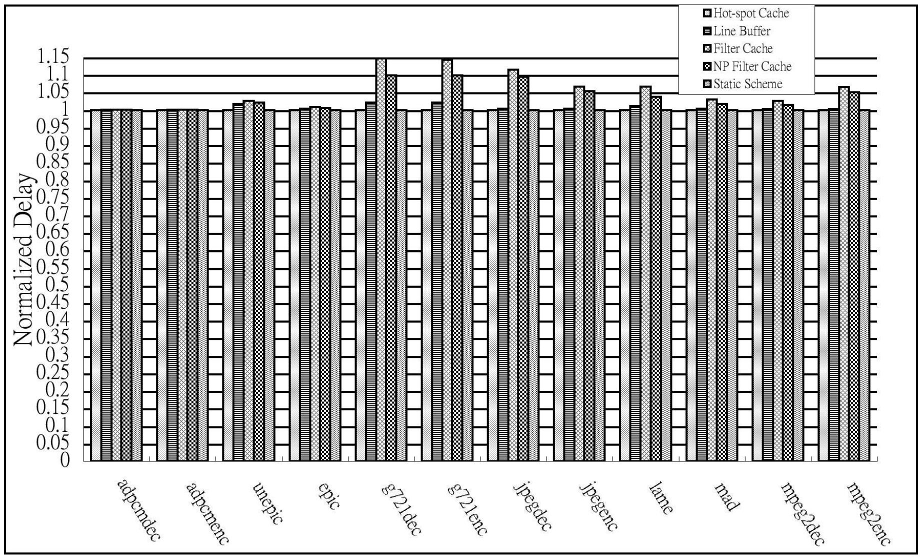

Figure 6.3: Normalized Delay of HotSpot Cache, Filter Cache, NP Filter Cache, and Static Scheme

## 6.3 Performance and Energy-Saving of the HotSpot Cache

Figure 6.2 shows the energy consumption (L0 cache + L1 cache) normalized to the base configuration (without L0 cache). We compare the hot-spot caches with other previously proposed mechanisms: one-entry line buffer, the filter cache, the next address predictive filter cache and the perfect static mechanism described in the previous section. The next address predictive filter cache proposed in [15] uses additional NP table to store the tag of next fetch address, and the next fetch address tag to compare current fetch address tag to predict next fetch mode. If the current fetch tag equals predicted next fetch tag, it is likely that the program is executing in a small loop that fits in the L0 cache and thus fetch from L0 cache; otherwise, fetch from L1 cache. This simple scheme relies on its prediction accuracy, and can only capture temporal reuses within small loops. The results show that the hot-spot cache achieve largest energy reduction compared with other schemes. The energy reduction from the line buffer are similar for all applications since it only utilizes the spatial locality within one cache block. The next address predictive filter cache does not work well for applications with large footprints, such as lame and mpeg2 encoder. The filter cache achieve energy reduction comparable to our scheme except for applications with high L0 cache miss rates, such as mpeg2 encoder and lame.

Figure 6.3 shows normalized execution time of the HotSpot Cache and other schemes. We can see that the normalized execution time of our proposed HotSpot Cache is close to 1 for all benchmarks. This shows that the performance guarantee is satisfied. Among all scheme tested, the hot-spot cache is the only one that can achieve energy savings with performance guarantee except for the line buffer.

# 7 Conclusion

In this paper, we propose an architectural approach to dynamically select basic blocks for storing 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. A mode controller is employed to determine which caches (L0 or L1) should be accessed during the instruction fetch stage. The proposed approach can achieve high L0 cache utilization without

[†] To determine the number of distinct phases for all tested applications, we perform the same branch behavior analysis of jpeg encoder shown in Figure 6.1 for all tested applications.

sacrificing performance. The simulation results show that the proposed mechanism can reduce the energy consumption of the instruction cache by 50% on the average for a set of multimedia applications without performance degradation.

## References

[1] J. Montanaro, 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

[2] J. Kin, M. Gupta, W. H. Mangione-Simith. The Filter Cache: An Energy Efficient Memory Structure. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, December, 1997

[3] 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*, Apr. 1995, pp. 63-68.

[4] *Intel StrongARM SA-1110 Microprocessor Brief Datasheet*, April 2000.

[5] 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.

[6] R. S. Bajwa, M. Hiraki. H. Kojima, D. Gorny, K. Nitta, A. Shridhar, K. Seki and K. Sasaki. Instruction Buffering to Reduce Power in Processors for Signal Processing. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 5, No. 4, December 1997.

[7] N. E. Bellas, I. N. Hajj and C. D. Polychronopoulos. Using Dynamic Cache Management Techniques to Reduce Energy in General Purpose Processors. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 8, No. 6, December 2000.

[8] N. E. Bellas, I. N. Hajj, C. D. Polychronopoulos and G. Stamoulis. Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 8, No. 3, June 2000.

[9] 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*, May, 1999, pp. 136-147.

[10] 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*, August 1999, pp. 63-68.

[11] 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.

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

[13] 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.

[14] Kanad Ghose and Milind B. Kamble Reducing Power In Superscalar Processor Caches Using Subbanking, Multiple Line Buffers And Bit-Line Segmentation. In *ISLPED 99*, San Diego, CA, USA, 1999.

[15] Weiyu Tang, Rajesh Gupta, and Alexandru Nicolau Design of a Predictive Filter Cache for Energy Savings in High Performance Processor Architectures. In *InternationalConference on ComputerDesign(ICCD)*, Austin, Texas, USA, 2001.

[16] J. S. Hu, A. Nadgir, N. Vijaykrishnan, M. J. Irwin, 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.