Exploiting Program Hotspots and Code Sequentiality for Instruction Cache Leakage Management

J. S. Hu, A. Nadgir, N. Vijaykrishnan, M. J. Irwin, M. Kandemir
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802, USA

mdl@cse.psu.edu

ABSTRACT

Leakage energy optimization for caches has been the target of much recent effort. In this work, we focus on instruction caches and tailor two techniques that exploit the two major factors that shape the instruction access behavior, namely, hotspot execution and sequentiality. First, we adopt a hotspot detection mechanism by profiling the branch behavior at runtime and utilize this to implement a HotSpot based Leakage Management (HSLM) mechanism. Second, we exploit code sequentiality in implementing a Just-In-Time Activation (JITA) that transitions cache lines to active mode just before they are accessed. We utilize the recently proposed drowsy cache that dynamically scales voltages for leakage reduction and implement various schemes that use different combinations of HSLM and JITA. Our experimental evaluation using the SPEC2000 benchmark suite shows that instruction cache leakage energy consumption can be reduced by 63%, 49% and 29%, on the average, as compared to an unoptimized cache, a recently proposed hardware optimized cache, and a cache optimized using compiler, respectively. Further, we observe that these energy savings can be obtained without a significant impact on performance.

Categories and Subject Descriptors

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

General Terms

Design, Performance, Measurement

Keywords

Leakage power, Cache design

1. INTRODUCTION

Static energy consumption due to leakage current is an important concern in future technologies [3]. As the threshold voltage continues to scale and the number of transistors on the chip continues to increase, managing leakage current will become more and more important. As on-chip caches constitute a major portion of the processor's transistor budget, they account for a significant share of the leakage power consumption. Leakage is projected to account for 70% of the cache power budget in 70nm technology [9].

Since the leakage current is a function of the supply voltage and the threshold voltage, it can be controlled by either reducing the supply voltage or by increasing the threshold voltage [12, 8, 6, 7, 1]. In this work, we utilize a drowsy cache [6] based on a multiplexed supply voltage as our circuit primitive for controlling leakage. In this circuit, in order to reduce leakage current, the supply voltage to the cache line is switched to the lowest level that can still retain the data. However, a cache line cannot be accessed when the supply voltage is lowered and the voltage must be switched back to normal value before the cache line access.

Here, we focus on reducing leakage energy in the instruction cache. A good leakage management scheme needs to balance appropriately the energy penalty of leakage incurred in keeping a cache line turned on after its current use with that of turning the line off (we use turn-off here to refer to a transition to the drowsy state, and turn-on to refer to waking up to normal active state.) and incurring the transition energy (for turning on a cache line) and performance loss that will be incurred if and when that cache line is accessed again. Our leakage management premise exploits two main characteristics of instruction access patterns to strike this balance: program execution is mainly confined in program hotspots and instructions exhibit a sequential pattern. A significant part of an application execution is spent in specific program hotspots (identification of hotspots is explained Section 3). We find this percentage of execution in hotspots to be 82% on the average for the SPEC2000 benchmark suite. In order to exploit this behavior, we propose a HotSpot based Leakage Management (HSLM) approach that is used in two different ways. First, it is used for detecting and protecting cache lines containing program hotspots from inadvertent turn-off. Second, HSLM can be used to detect a shift in program hotspot and to turn off cache lines closer to their last use. Next, we present a Just-in-Time Activation (JITA) scheme that exploits sequential access patterns for instruction caches by predictively activating the next cache line when the current cache line is accessed. Employing both HSLM and JITA in conjunction with previous approaches, our experimental evaluation using SPEC2000 benchmarks shows that it provides 49% and 29% more leakage energy savings in the instruction cache as compared to schemes in [9] and [13].

The rest of this paper is organized as follows. Section 2 provides a more detailed view of factors influencing leakage reduction and how our approach relates to existing schemes. Section 3 details the implementation of our HSLM and JITA strategies. Section 4 explores different leakage management approaches that combine HSLM and JITA. An experimental evaluation of the different schemes is performed in Section 5. Finally, we provide conclusions in Section 6.

2. RELATED WORK

Previous approaches for leakage energy management utilize runtime information to make leakage control decisions. DRI I-Cache [12] dynamically resizes the instruction cache and puts the disabled portion of the cache into leakage-control mode by monitoring the cache miss rate. This scheme is coarse-grain in managing leakage as it turns off large portions (by halfing) of the cache depending on a performance feedback that does not specifically capture cache line usage patterns.

Drowsy cache proposed in [6] periodically turns off cache lines independent of the instruction access pattern. The success of this strategy depends on how well the selected period reflects the rate at which the instruction working set changes. On the plus side, this approach is simple and has very little implementation overhead. Technique proposed in [9] adopts a bank based strategy, where as execution moves from one bank to another, the hardware turns off the former and turns on the latter. Two severe problems associated with this scheme, namely, unnecessarily turning on cache lines that will never be touched and keeping these cache lines in active mode until the next bank transition result in poor energy behavior. Frequent bank transitions due to a small loop mapping across two banks can also lead to inferior energy and performance behavior.

The cache decay mechanism proposed by Kaxiras et al. in [8] shuts down cache lines if they are not accessed for some certain number of cycles (decay interval). In fact, the problems associated with selecting a good decay interval are similar to those associated with selecting a suitable turn-off period in [6]. The adaptive version of the cache decay scheme [8] tailors the decay interval for the cache lines based on cache line access patterns. The approach in [14] also uses tag information to adapt leakage management.

In [13], an optimizing compiler is used to analyze the program to insert explicit cache line turn-off instructions. This scheme demands sophisticated program analysis and modifications in the ISA to implement cache line turn-on/off instructions. In addition, this approach is only applicable when the source code of the application being optimized is available.

Another important limitation of existing leakage control schemes is that most of the techniques only focus on a turn-off mechanism and activate turned-off cache lines (or banks) only when accessed, which can lead to performance penalties. A notable exception to this is the predictive bank turn-on scheme employed in [9].

3. USING HOTSPOTS AND SEQUENTIALITY IN MANAGING LEAKAGE

Having analyzed the shortcomings of directly applying existing approaches to instruction cache leakage management, our goal is to support a turn-off scheme that is sensitive to program behavior changes and that captures both temporal and spatial locality variances. Further, we would like a predictive turn-on mechanism to support the sequentiality of instruction cache accesses. However, we want to keep the granularity of predictive turn-on as small as possible so that the cache lines are turned on if and only if they are needed.

In this paper, we propose two mechanisms to support leakage management of instruction caches. First, we propose a HotSpot based Leakage Management scheme (HSLM) that tracks program behavior. Second, we propose a Just-in-Time Activation (JITA) scheme for turning on the next cache line to exploit the sequentiality of code accesses.

3.1 HSLM: HotSpot Based Leakage Management

Previous research shows that a program execution typically occurs in *phases* [11]. Each phase can be identified by a set of instructions that exhibit high temporal locality during the course of execution of the phase. Two important observations made by previous research are that phases can share instructions and that the instructions in a given phase do not need to be tightly clustered together in one portion of the address space. In fact, they can be scattered all over the address space as pointed out in [11]. Typically, when execution enters a new phase, it spends a certain number of cycles in it. When this number is high, one can refer to that phase as a *hotspot.* Since branch behavior is an important factor in shaping the instruction access behavior, we track the hotspots using a branch predictor. While the use of branch predictors for optimizing programs has been used in the past (e.g., see [11]), to our knowledge, this is the first study that employs branch predictors for improving cache leakage consumption.

Detecting program hotspots can bring two main advantages. First, we can know which cache lines are going to be the most active ones and prevent them from being turned off. Second, we can turn off the cache lines that hold instructions that do not belong to a newly detected hotspot.

3.1.1 Protecting Program Hotspots

Our leakage management approach builds on the drowsy cache technique [6] that periodically transitions all cache lines to drowsy mode by issuing a global turn-off signal which sets register Q of leakage control circuitry in Figure 1. A global (modulo-N) counter is used to control the periodic turn-off. In order to protect the cache lines containing the program hotspots from inadvertent turn-off, we augment each drowsy cache circuit with a local *voltage control mask bit (VCM).* If this mask bit is set, the corresponding cache line will mask the influence of the global turn-off signal and prevent turn-off. In order to identify execution within hotspots, we augment and utilize the information from the branch target buffer (BTB) as explained in detail in the next paragraph. Once the program is identified to be within a program hotspot (or not), the *global mask bit (GM)* (Figure 2) is set (reset). When this global mask bit is set, the voltage control mask bit of all cache lines accessed is set to one to indicate that these cache lines form the program hotspot.

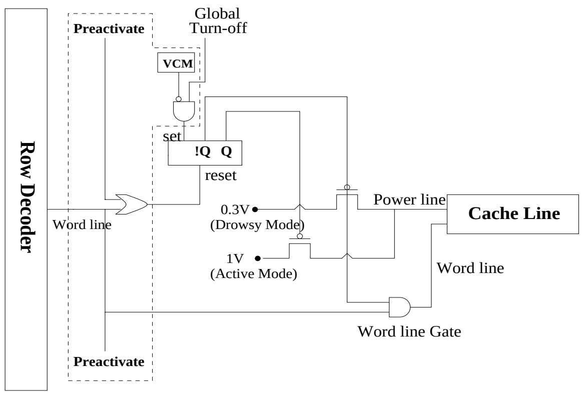

Figure 1: Leakage control circuitry supporting Just-in-Time Activation (JITA).

In a set-associative cache, the voltage control mask bit is set based on the tag match results of the cache access and is performed only for the way that actually services the request. The voltage control mask bits are reset on cache line replacements.

Our hotspot detection mechanism tracks the branch behavior information using the BTB. The BTB entries are augmented to collect the execution frequencies of basic blocks. Compared to the conventional BTB entry, the augmented structure includes three additional fields: the valid bit (vbit) for target address, an access counter for the target basic block (tgt_cnt), and an access counter for the fall-through basic block (fth_cnt). This new structure is shown in Figure 2. The valid bit indicates whether the current value of target address is valid or not. The valid bit is needed as a new entry can be added to the augmented BTB by both taken and non-taken branches. If the new entry is introduced when the branch is taken (not taken), the valid bit is set to one (zero). The access counter for the target (fall-through) basic block records how many times the branch is predicted as taken (not-taken). These counters are accessed and updated during each branch prediction according to the outcome of the prediction.

The value of the target/fall-through counter shows the frequency of the target/fall-through basic block fetched within a given sampling window and is compared with a predefined threshold T_{acc} to determine the hotness of the corresponding basic block. Each counter in the BTB has $\log(T_{acc}) + 1$ bits. The counters are initially set to zero when a new BTB entry is created. During a branch prediction, if the BTB hits, the corresponding counter is read out according to the outcome of the prediction and then incremented. Next, the most significant bit of the corresponding counter is checked to determine the hotness of the basic block starting at the target/fall-through address. If this bit is set, it means that the next (target or fall-through) basic block has exceeded the threshold T_{acc} number of accesses and subsequent fetches are part of a program hotspot. We set the global mask bit to capture this detection of a program hotspot. The global mask bit is reset when the most significant bit of the access counter for a subsequent BTB lookup is not set or when a BTB miss happens.

When a sampling window expires (determined by zeroing of the global counter), several initialization operations take place. First, a global turn-off signal is issued to turn off all cache lines except those with their voltage control mask bit set. Second, a global reset signal resets all voltage control mask bits. This is performed to track variances in program behavior hotness. Third, all the access counters in the BTB are shifted right by one bit to reduce their access count by half. This is performed to reduce the weight for accesses performed in an earlier period when determining hotness. Subsequently, a new sampling window begins and the operations repeat.

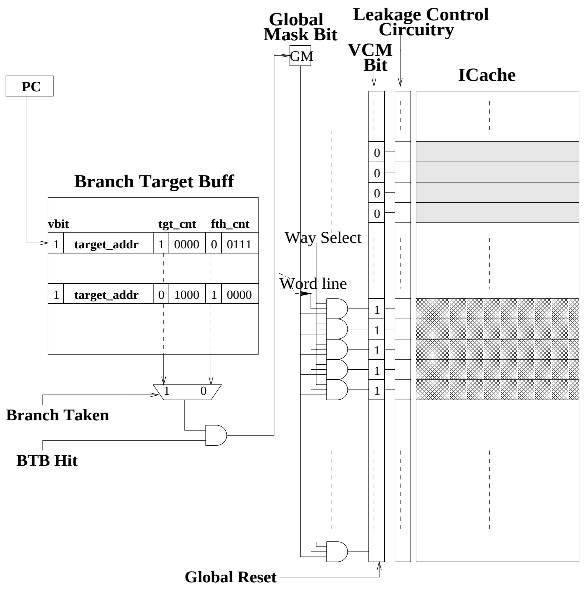

Figure 2: Microarchitecture for HotSpot based Leakage Management (HSLM) scheme. Note that O/P from AND gates go to the set I/P of the mask latches.

3.1.2 Detecting New Program Hotspots

One of the drawbacks of periodic approaches is that cache lines can be turned off only when a preset period expires. It would be more beneficial if older (not recently accessed) cache lines can be turned off immediately when a shift in hotspot is detected. Our approach is specifically targeted at identifying a shift of the program hotspot to a new loop. Specifically, if the target counter in the BTB entry of a predicted taken branch indicates that the target basic block is in a hotspot (the most significant bit of the counter is "1") and if the target address is lower than the current program counter value, we assume that the program is in a hotspot executing a loop. At this point, the global turn-off signal is issued and all cache lines except those corresponding to hotspots are switched to drowsy mode. In the schemes evaluated in this paper, we always use a periodic turn-off issued when the global counter expires in addition to the dynamic loop-based turnoff to account for the cases where the execution remains within the same loop for a long time or when there are few loop constructs.

3.2 JITA: Just-In-Time Activation

In many applications, sequentiality is the norm in the code execution. In addition, optimizations such as loop unrolling, superblock and hyperblock formation increase the sequen-

Schemes	Turn-off Mechanism	Turn-on Mechanism	Turn-on Gran.	Turn-off Gran.
Base	-	When accessed	Cache line	-
Drowsy-Bank	Switch banks	Bank prediction	Bank	Bank
Loop	Instruction	When accessed	Cache line	Entire cache
DHS	Periodic+Hot backward branch+Not Hot	When accessed	Cache line	Entire cache
DHS-PA	Periodic+Hot backward branch+Not Hot	When previous line is accessed	Cache line	Entire cache
DHS-Bank-PA	Periodic+Hot backward branch+Switch banks+Not Hot	When previous line is accessed	Cache line	Entire cache

Table 1: Leakage control schemes evaluated.

tiality of the code [4, 5, 10]. The sequential nature of code can be exploited to predict the next cache line that will be accessed and mask the penalty for transitioning a cache line from drowsy to active mode just-in-time for access. Specifically, we propose a scheme that preactivates the next cache line, JITA.

The leakage control circuitry augmented to support JITA for a direct-mapped cache is illustrated in Figure 1. When the current cache line is being accessed, the voltage control bit for the next cache line (next index) is reset, thereby transitioning it to the active state. Thus, when the next fetch cycle occurs and there is code sequentiality, the next required cache line is already in the active mode and ready for access. However, the JITA scheme is not successful when a taken branch occurs and the target is beyond the next cache line or when the next address falls in a different memory bank. While the same circuit can be employed for a set-associative cache, it would lead to activating the cache lines in all the ways of the same set. In order to avoid this, we utilize way prediction information associated with the next cache line to activate only the cache line of a selected way. This scheme is found to work well as programs expend a major part of their time in their hotspots.

4. DESIGN SPACE EXPLORATION

Table 1 shows the different cache leakage savings approaches that we evaluated. In all cases, we assume a cache line is in drowsy mode before its first access.

The `Drowsy-Bank` scheme [9] employs a turn-off policy that is based on the assumption that bank access changes indicate a shift in locality. The `Loop` scheme (the most aggressive scheme in [13]) turns off all cache lines after executing each loop. These two schemes are used for comparison.

In the `DHS` (Dynamic HSLM) scheme, the global turn-off signal is issued when a new loop-based hotspot is detected. Thus this scheme can turn off unused cache lines before the fixed period is reached by detecting that execution will remain in the new loop based hotspot. This scheme also employs the hotspot detection for protecting cache lines containing program hotspots. The `DHS` scheme also incurs a penalty due to the masking that can delay the turn-off of cache lines that belonged to an older hotspot until the identification of a new hotspot or the expiration of an additional period as compared to a periodic scheme that employs no masking. The `DHS-PA` scheme employs the JITA strategy on top of the `DHS` scheme.

Our final approach, `DHS-Bank-PA` issues the global turn-off signal at fixed periods, when the execution shifts to a new bank, or when a new loop hotspot is detected. It attempts to identify both spatial and temporal locality changes. Further, it employs the mask bits set using hotspot detection to protect active cache lines and the JITA scheme for predictive cache line turn-on.

Predictive turn-on strategies are not without their drawbacks. When a wrong prediction is made, they not only incur a performance penalty to turn on the needed line (also associated with techniques that have no prediction) but also the energy cost incurred in activating the wrong cache line.

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the leakage control schemes described in the previous section. First, we describe our simulation parameters. Next, we compare the energy, performance and energy-delay results of the different schemes.

5.1 Experiment Setup

We extended SimpleScalar 3.0 [2] to implement the schemes presented in Table 1. We simulate a contemporary microprocessor similar to Alpha 21264. The base configuration parameters of the processor and memory hierarchy are given in Table 2. We use a set of ten integer and four floating point applications from the SPEC2000 benchmark suite and use their PISA binaries and reference inputs for execution. Each benchmark is first fast-forwarded half a billion instructions, and then simulated the next half a billion committed instructions. Table 3 gives the technology and energy parameters used in this paper. The energy parameters are based on a 70nm/1.0V technology [6].

Processor Core	
Instruction Window	64 RUU, 32 LSQ
Decode/Issue Width	4 instructions per cycle
Commit Width	4 instructions per cycle
Function Units	4 IALU, 1 IMULT/IDIV, 4 FALU, 1 FMULT/FDIV, 2 Memports
Branch Predictor	Bimodal, 2048 entries, 512-set 4-way BTB, 8-entry RAS
Memory Hierarchy	
L1 ICache	32KB, 1 way, 32B blocks, 1 cycle latency
L1 DCache	32KB, 4 ways, 32B blocks, 1 cycle latency
L2 UCache	256KB, 4 ways, 64B blocks, 8 cycle latency
Memory	80 cycles first chunk, 8 cycles rest
TLB	4 way, ITLB 64 entry, DTLB 128 entry, 30 cycle miss penalty

Table 2: Configuration parameters for the simulated processor and its memory hierarchy.

Our energy model is as follows:

$$E_{energy} = E_{drowsy} + E_{active} + E_{datapath+dcache} + E_{overhead}.$$

$$E_{overhead} = E_{turnon} + E_{extraturnon} + E_{btbcounters} + E_{misc}.$$

The total effective leakage energy E_{energy} of the instruction cache with leakage management schemes is composed of four parts: leakage energy E_{drowsy} consumed by the cache lines in drowsy mode, leakage energy E_{active} consumed by cache lines in active mode, the increased leakage energy consumption in datapath and data cache $E_{datapath+dcache}$ due

Technology and Energy Parameters	
Feature Size	70nm
Threshold Voltage	0.2V
Supply Voltage	1.0V
Clock Speed	1.0GHz
L1 cache line Leakage in Active	0.417pJ/cycle
L1 cache line Leakage in Drowsy	0.0663pJ/cycle
Transition (drowsy to active) Energy	25.6pJ
Transition (drowsy to active) latency	1 cycle
Dynamic Energy per BTB counter (5 bits)	0.96pJ/transaction
Simulation Parameters	
Window Size	2048 cycles
Hotness Threshold (T_{acc})	16
Subbank Size	4K Bytes

Table 3: Technology and energy parameters for the simulated processor given in Table 2.

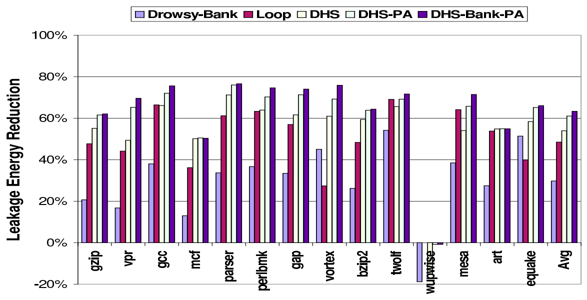

Figure 3: Leakage energy reduction w.r.t the `Base` scheme.

to the increased cycles incurred by leakage control, and the overhead energy $E_{overhead}$ for implementing the leakage control schemes. The overhead energy $E_{overhead}$ includes transition energy E_{turnon} for waking up a drowsy cache line, extra transition energy $E_{extraturnon}$ due to unnecessary turn-ons resulting from predictive cache line turn-on schemes, the dynamic energy $E_{btbcounters}$ consumed in BTB counters introduced for HSLM, and miscellaneous energy consumption E_{misc} due to voltage control mask bits and a way predictor, if used, in set-associative cache. Since the BTB counters have a very high percentage of zero bits (an average of 95%, due to saturation or not being touched), these counters are implemented using asymmetric-Vt SRAM cells [1]. The optimized cells consume only 1/10th of the original leakage when storing zeros. Thus, additional leakage due to BTB counters is very small.

5.2 Experimental Results

Figure 3 presents the total leakage energy reduction of all leakage control schemes compared to the `Base` scheme. This evaluation depends on the overhead leakage incurred in the rest of the chip excluding the instruction cache. In order to capture different processor configurations and underlying circuit styles, we vary the instruction cache leakage from 10-30% of overall on-chip leakage. We observed that `DHS-Bank-PA` has the best energy behavior for all values experimented in this range as HSLM and JITA help to reduce additional overhead energy for this scheme. Specifically, it achieves an average energy reduction of 63% over `Base`, 49% over `Drowsy-Bank`, and 29% over `Loop` when instruction cache accounts for 30% of on-chip leakage. When this percentage is 10%, these energy reductions are 59% over `Base`, 44% over `Drowsy-Bank`, and 50% over `loop` (Not shown in figure for brevity). Focusing on an anomalous trend in Figure 3, benchmark *wupwise* exhibits very different energy behavior. Except for scheme `Loop` (0% reduction), all other schemes increase the energy consumption with the `Drowsy-Bank` scheme increasing energy consumption by 19%. This results from the small instruction footprint of this benchmark during simulation which touches only 77 cache lines of the same bank (out of 128 lines for our configuration).

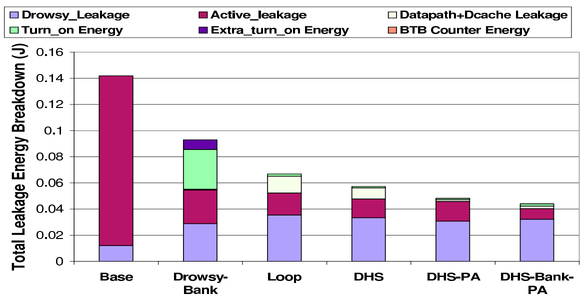

Figure 4: The leakage energy breakdown (average over fourteen SPEC2000 benchmarks..

In order to have a closer look at the energy behavior of the different schemes, Figure 4 provides a more detailed breakdown (averaged over all benchmarks). For `Base`, the leakage energy is due to leakage energy consumed by drowsy cache lines (before a cache line is accessed) and leakage energy consumed by active cache lines. `Loop` and `DHS` have a noticeable portion of energy from additional datapath and data cache leakage energy due to the performance degradation. In contrast, `Drowsy-Bank` has very little performance overhead, because it predictively turns on banks. However, turn-on energy for the `Drowsy-Bank` scheme is significant, as it turns all lines in a bank in one cycle. Further, the extra turn-on energy (i.e., due to activating the wrong subbank) is around 20% of the overall turn-on overhead energy and 8% for the total leakage energy. Further, we notice that the BTB counter overhead is very low since the saturated counters do not incur any additional dynamic activity as their clocks are gated once the most significant bit turns to a one (until it is reset). Only the most significant bit of these saturated counters is used to identify hotspots.

We also measured a metric defined as the *turn-on-hit* ratio to highlight the performance penalty for accessing drowsy cache lines. This ratio provides the percentage of cache line activations on cache access hits to the total number of activations performed. A larger value indicates more performance penalty. Activation on cache misses does not incur any additional performance penalty. For `Loop` and `DHS`, this value is 79.5% and 87%, on average respectively. The use of JITA reduces this ratio to 11% and 12.4% for `DHS-PA` and `DHS-Bank-PA`. While JITA is successful in reducing the penalty of activation, it still incurs some penalty when it fails due to taken branches or jumps to drowsy cache lines.

Figure 5, shows how this activation penalty translates into actual performance values. The `Base` scheme (not shown) performs the best as it incurs no performance penalties except for initial activation of untouched cache lines. The `Drowsy-Bank` scheme performs the best among other schemes and incurs only a degradation of 0.56%. The `Loop` scheme incurs the highest degradation of 15.4% degradation on the

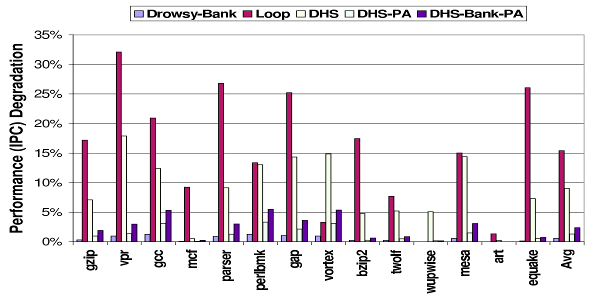

Figure 5: Performance (IPC) degradation w.r.t the `Base` scheme.

average because this scheme had the highest number of accesses to drowsy cache lines on average (due to the absence of predictive activation or masking for hotspots). The best performing energy scheme, `DHS-Bank-PA` suffers a degradation of 2.3% on the average.

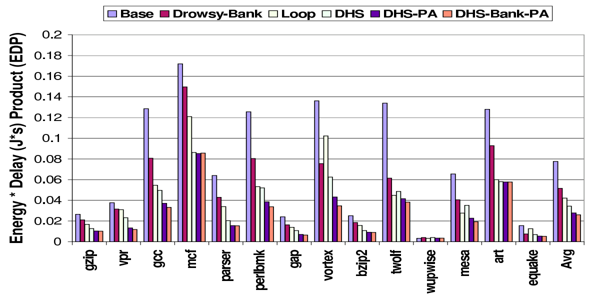

Figure 6: Energy delay (J*s) product (EDP).

Finally, we present the energy delay products (EDP) of each scheme in Figure 6. We have included the overhead energy (see Figure 4). The results show that our scheme `DHS-Bank-PA` performs best. It achieves the smallest EDP value, which has an average reduction of 62.63% over `Base`, and additionally 48.3% and 37.7% over `Drowsy-Bank` and `Loop` respectively. Experiments with different cache configurations (in cache size, number of sets, associativity, and cache line size) also show that `DHS-Bank-PA` performs best (detailed results are not shown due to the space limitation).

6. CONCLUSIONS

This work focused on the leakage management of instruction caches. Our leakage management exploits two main characteristics of instruction access patterns: that program execution is mainly confined in program hotspots and that instructions exhibit a sequential access pattern. We devise two strategies: a HotSpot based Leakage Management (HSLM) and Just-in-Time Activation (JITA) to exploit these two main characteristics.

Specifically, we used HSLM to protect turning off cache lines containing program hotspots and for dynamically identifying shifts in program hotspot. JITA was used to predictively activate the next cache line to mitigate the performance penalty incurred in waking up drowsy cache lines. We find that using program behavior captured by HSLM helps avoid some of the overheads of managing leakage in an application agnostic fashion and also helps to detect shifts in program hotspots dynamically. Further, we find that JITA is a simple and effective scheme for masking the performance penalties associated with waking up drowsy cache lines and permits a fine-grain leakage management at the cache line level. With the increasing focus on reducing leakage energy as technology scales and the incorporation of larger and larger caches on-chip, such cache leakage control schemes will be vital in future processor generations.

7. ACKNOWLEDGMENTS

This work was supported in part by a grant from MARCO 98-DF-600 GSRC, NSF CAREER Awards 0093082 and 0093085, and NSF grants 0082064 and 0103583.

8. REFERENCES

[1] N. Azizi, A. Moshovos, and F. N. Najm. Low-leakage asymmetric-cell sram. In *Proc. the 2002 International Symposium on Low Power Electronics and Design*, Monterey, CA, 2002.

[2] 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, April 2000.

[3] J. A. Butts and G. Sohi. A static power model for architects. In *Proc. the 33th Annual International Symposium on Microarchitecture*, December 2000.

[4] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proc. the 29th Annual Hawaii International Conference on System Sciences*, pages 183–192, Maui, HI, January 1996.

[5] P. P. Chang et al. Three superblock scheduling models for superscalar and superpipelined processors. Technical Report CRHC-91-29, Center for Reliable and High-Performance Computing, University of Illinois, Urbana, IL, 1991.

[6] K. Flautner et al. Drowsy caches: Simple techniques for reducing leakage power. In *Proc. the 29th International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[7] S. Heo et al. Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *Proc. the 29th International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[8] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. the 28th International Symposium on Computer Architecture*, Sweden, 2001.

[9] N. Kim, K. Flautner, D. Blaauw, and T. Mudge. Drowsy instruction caches: Leakage power reduction using dynamic voltage scaling and cache sub-bank prediction. In *Proc. the 35th Annual International Symposium on Microarchitecture*, November 2002.

[10] S. A. Mahlke et al. Effective compiler support for predicate execution using the hyperblock. In *Proc. the 25th Annual International Symposium on Microarchitecture*, 1992.

[11] M. C. Merten et al. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[12] M. D. Powell et al. Reducing leakage in a high-performance deep-submicron instruction cache. *IEEE Transactions on VLSI*, 9(1), February 2001.

[13] W. Zhang et al. Compiler-directed instruction cache leakage optimization. In *Proc. the 35th Annual International Symposium on Microarchitecture*, November 2002.

[14] H. Zhou et al. Adaptive mode control: a static power-efficient cache design. In *Proc. the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001.