2013

Low Cost Dynamic Architecture Adaptation Schemes for Drowsy Cache Management

Nitin Prakash
University of Massachusetts - Amherst, nitinprakash.7@gmail.com

LOW COST DYNAMIC ARCHITECTURE ADAPTATION SCHEMES FOR DROWSY CACHE MANAGEMENT

A Thesis Presented

by

NITIN PRAKASH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

February 2013

ELECTRICAL AND COMPUTER ENGINEERING

LOW COST DYNAMIC ARCHITECTURE ADAPTATION SCHEMES FOR DROWSY CACHE MANAGEMENT

A Thesis Presented

by

NITIN PRAKASH

Approved as to style and content by:

Israel Koren, Co-Chair

C. Mani Krishna, Co-Chair

Wayne P. Burleson, Member

Christopher V. Hollot, Department Head
Electrical and Computer Engineering

ACKNOWLEDGMENTS

I would like to thank NSF for funding this project, and the Architecture and Real Time Systems (ARTS) Laboratory at the University of Massachusetts, Amherst for allowing me to work on it.

I am deeply grateful to Prof. Israel Koren and Prof. Mani Krishna for their invaluable guidance and help throughout the course of this project. I would like to express my gratitude to Prof. Wayne Burleson for his valuable time, and for accepting to be on my thesis committee. His ideas have gone a long way in improving the quality of this work.

Finally, I extend my heartfelt thanks to my family and friends who have supported me throughout my time here at the University.

ABSTRACT

LOW COST DYNAMIC ARCHITECTURE ADAPTATION SCHEMES FOR DROWSY CACHE MANAGEMENT

FEBRUARY 2013

NITIN PRAKASH

B.E., MANIPAL UNIVERSITY, MANIPAL, INDIA
M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professors Israel Koren and C. Mani Krishna

Energy consumption and speed of execution have long been recognized as conflicting requirements for processor design. In this work, we have developed a low-cost dynamic architecture adaptation scheme to save leakage power in caches. This design uses voltage scaling to implement *drowsy caches*. The importance of a *dynamic* scheme for managing drowsy caches, arises from the fact that not only does cache behavior change from one application to the next, but also during different phases of execution within the same application. We discuss various implementations of our scheme that provide a tradeoff between granularity of control and design complexity.

We investigate a combination of policies where the cache lines can be turned off completely if they are not accessed, when in the drowsy mode. We also develop a simple dynamic cache-way shutdown mechanism, and propose a combination of our dynamic scheme for drowsy lines, with the cache-way shutdown scheme. Switching off cache ways has the potential of greater energy benefits but provides a very coarse grained control. Combining this with the fine grained scheme of drowsy cache lines allows us to exploit more possibilities for energy benefits without incurring a significant degradation in performance.

Keywords: Drowsy Cache, Architecture Adaptation, Low Power, Leakage Reduction, Dynamic Scheme

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1
INTRODUCTION

1.1 Overview

Computing platforms have been facing the dilemma of energy consumption versus performance requirements, for a long time. The trade-off is especially important for real-time systems used in situations where energy is scarce and/or heat dissipation is costly. Increased power dissipation also leads to an increase in the temperature, which directly affects the reliability of the system. As reported in [5], the delay fault rate doubles for every 10°C increase in the operating temperature. A 10-15°C increase in the temperature can halve the life span of a circuit [15].

Traditionally, Dynamic Voltage and Frequency Scaling (DVFS) has been used for reducing energy consumption at the expense of execution speed. Researchers have studied the complementary approach of Architecture Adaptation, which takes advantage of the application specific behavior and turns off unused sections of the hardware. As modern embedded systems use more and more complex hardware for their plethora of applications, the scope for architecture adaptation increases. But, so does the complexity of finding the optimum configurations, in the face of a variety of applications with different demands on the hardware.

With considerable increases in the size of on-chip memory, and the steady decrease in the feature size of transistors, leakage power has become a major contributor to the total energy consumption in processors [8]. Furthermore, leakage power increases exponentially with a rise in operating temperature, which in turn increases the temperature even more rapidly. Most of this leakage power originates in the caches, due to the presence of a large number of transistors. We therefore, target in this work the on-chip cache units and attempt to reduce the performance penalty while still achieving lower energy consumption.

1.1 Contributions of this work

As real-time applications become ever more complex with varying demands on caches, there is a need for *dynamic* architecture adaptation policies that work across a wide range of applications with minimal tuning requirements. In this work, we have implemented such a dynamic scheme for drowsy cache management. *Drowsy cache* is a scheme where a cache line is put to a low voltage mode, which consumes less power and preserves the information on the cache line. However, to access the data, the cache line has to be reinstated to the high voltage mode. Typically, the cache lines are put to drowsy mode if there is no activity in a certain time interval. Instead of static selection of intervals, we propose a low overhead dynamic adaptation scheme, where the interval is selected based on the runtime behavior of the application. Unlike earlier methods (discussed in Chapter 2), our dynamic scheme uses performance counters to manage drowsy caches. This gives us a fine-grained control based on current application behavior. We show that our scheme is robust and can be used across a wide range of applications and cache configurations, without the need for re-tuning the algorithm parameters for every case.

The earlier methods that use this voltage scaling technique for drowsy caches (discussed in Chapter 2), assume that each row in the SRAM array has its own Vdd line. However, modern SRAM arrays share the Vdd contacts amongst adjacent rows. Our design takes this into account and controls a pair of cache lines using a single voltage controller. This greatly reduces the associated design and area overhead, while providing a similar granularity of control as with dedicated Vdd lines. To the best of our knowledge, this is the first attempt to implement a scheme of drowsy lines by taking into account shared Vdd contacts.

Furthermore, we investigate a combination of the drowsy cache and the Gated-Vdd schemes. The Gated-Vdd scheme switches off cache lines completely and hence provides more leakage benefits, but at the risk of an increase in runtime due to increase in the number of cache misses.

We also propose a simple scheme for cache-way shutdown and integrate it with our previous design of drowsy cache lines. The cache-way policy provides a coarse-grained control by switching off entire cache ways. The drowsy line policy provides finer control as it targets individual cache lines, and puts them into a low-voltage mode when not in use. We show that this integrated approach provides more energy benefits than previously known designs while still maintaining minimal performance degradation.

CHAPTER 2
REVIEW OF RELATED WORK

2.1 Reducing leakage power in cache lines

Various designs such as gated-Vdd [12], ABB-MTCMOS [10] (dynamically increasing threshold voltage), and voltage scaling have been proposed to control leakage power in transistors. Kaxiras et al. [6] have used the gated-Vdd method along with counters to control the drowsy cache lines. Miss rate is used to determine an optimal value for the counters: once they saturate, the corresponding cache lines are placed in drowsy mode. In the gated-Vdd scheme, the memory cell loses its data and hence, a hit on a drowsy line causes a miss. Similarly, Powell et al. [13], using the gated-Vdd method, have proposed a mechanism to identify an application's I-cache requirements in order to reduce the leakage current. Using a threshold scheme, the mechanism reacts to changes in miss rate by changing the number of sets in the cache. The proposed mechanism uses a variable set mask to properly access the corresponding set.

Flautner, et al. [3] have shown a circuit implementation for drowsy cache lines using voltage scaling. The supply voltage to the SRAM cell is scaled down, to around 1.5 times the threshold voltage. The sub-threshold leakage due to short channel effects is significantly reduced. The authors have implemented a static scheme on a 32 KB cache using a *Simple Policy*, in which they select an interval, and put to drowsy mode, all cache lines, at the end of the interval. Only a single global counter is required for this scheme. A wakeup cost is incurred only on the currently active footprint of the cache. They have discussed a *NoAccess Policy*, where only the non-accessed lines in the specified interval are put to drowsy mode. Based on their Simple Policy and other considerations like wakeup transition time and processor architecture, they show that a static interval of 2000 to 8000 cycles works adequately for their benchmarks. They also point out the fact that their simple algorithm does not work well for the Icache. The authors in their next work [7] have proposed a cache sub-bank prediction technique for Icache. The cache is divided into sub-banks and only one sub-bank is awake at any given time. A sub-bank prediction buffer is included that stores the instructions (which lead to a change in the sub-bank) and the address of the next predicted sub-bank. This method has increased area and dynamic power overheads. An alternate approach is presented, in which the predicted address is included in the tag array. But this information is lost when the cache line is replaced, and hence is not suitable for applications where miss rates are high.

Geiger et al. [4] have proposed a combination of drowsy cache lines (with static intervals) and region-based caches. Petit et al. [11] rely on the reuse information to control drowsy caches. This mechanism is used for set-associative caches, wherein they keep awake only the most recently used line in every set. A comparison is made with the scheme where the two most recently used lines are kept awake in the set. They have also proposed a combination of the two, to get a balance between energy benefits and performance degradation. The benefit of this method depends on the associativity of the cache, and it cannot be used in a direct-mapped or a fully associative cache.

Zushi et al. [32] have proposed an improvement on the MRU scheme. In addition to the MRU information they record access information for every cache line at regular (predetermined) intervals. A global drowsy update signal is activated at the end of every interval. The lines are put to drowsy mode based on the MRU bit and the access information of the last interval. We present a detailed comparison to various flavors of this scheme in Section 6.6.2.

Alioto et al. [31] have proposed a scheme to exploit locality wherein they put active lines to drowsy mode immediately after the access moves on to another line. They have devised localized control, based on the observation that the newly accessed cache lines are nearby to the previously accessed line (spatial locality). This scheme works well for sequential codes but degrades quickly if the number of branches is high. We present a detailed comparison to this scheme in Section 6.6.3.

Recently, researchers have proposed working at the granularity of sub-blocks within a cache line. This is based on the observation that all the words in a cache line may not be accessed, especially if the cache line is large. Chen et al. [33] have proposed a prediction scheme where they predict the access pattern of every sub block in a cache line. Extra bits are included in the tag array (one for every sub-block), which store information regarding which sub-blocks were accessed. This information is then transferred to a Pattern History Table when the cache line is replaced. Next time the line is fetched, the sub-blocks that are predicted not be accessed are switched-off (using the Gated Vdd scheme). Alvez et al. [34] have extended and modified the design to predict when the accessed sub-blocks become dead. To implement this, they have included 2-bit usage counters for every sub-block (instead of a single bit). They also include an overflow bit to indicate that the predicted accesses are more than what the counters can contain. These sub-blocks are switched-off after the predicted number of accesses. If the overflow bit is set, then the sub-block is never switched off. This scheme has a fairly high area overhead. They have reported the size of the additional structures to be 6.1 KB for implementing this scheme on a 32 KB L1 cache. Furthermore, extra control signals are required by the Gated-Vdd scheme to implement it at a sub-block level.

The authors of [7][27][28] have taken an orthogonal approach to decreasing the performance degradation caused by drowsy caches. Instead of optimally controlling the time when the cache lines should be put to drowsy mode, they try to predict future accesses and wake up the line before the access is done. These schemes put all lines to drowsy mode at predetermined static intervals. Then, prediction information is used to predict future accesses and wake up the lines. [27] proposes a DHS (Dynamic HotSpot based leakage reduction) policy, which uses the BTB (Branch Target Buffer) to detect loops. A global drowsy signal is issued when a new loop based hotspot is detected. On top of DHS, it employs the JITA (Just In Time Access) policy that awakes up the next sequential line when an access is made. [28] directly uses the branch predictor information to predict the next line to be woken up. To hide the latency, an extra pipeline cycle is introduced between the branch predictor access and instruction fetch stage. This method incurs a penalty in the event of a branch target misprediction. These prediction-based methods can be used on top of our dynamic adaptation scheme.

2.2 Other cache reconfiguration schemes

Various authors have explored different schemes to configure caches based on application requirements, in order to achieve energy savings. Cache configuration is done mainly by switching off cache ways, sets, or changing the associativity. Others approaches include some modification in the cache lookup schemes to reduce the energy expended.

The phased-lookup cache [21] uses a two-phase lookup, where all tag arrays are accessed in the first phase, and upon a hit, only one data way is accessed in the second phase, resulting in less energy at the expense of longer access time. Way predictive set-associative caches [20][22][24] access one tag and data array initially (based on the prediction), and only access the other arrays if that initial array did not result in a match, again resulting in lower energy consumption at the expense of longer average access time. In [22], the authors have used an MRU (Most Recently Used) scheme for prediction. To hide the latency of prediction, the set-index address is calculated at an earlier stage in the pipeline. In [24], the authors have used a lookup table based predictive scheme. They have studied the effectiveness of the predictive scheme and selective direct-mapped caches [19] for Icache and Dcache. Filter caching [23] introduces a small (and hence low-power) direct-mapped cache in front of the regular cache. If most of a program's time is spent in small loops, then most hits would occur in the filter cache, thus reducing overall energy consumption.

Albonesi [17] has proposed a scheme to disable cache ways to save dynamic power. Based on the allowed Performance Degradation Threshold (PDT), the applications are profiled to find the optimum number of cache ways that can be disabled. A simple AND gate structure is used to disable the access to particular cache ways. He argues that different PDT values can be used for different instantiations of the same application, and that the operating system or a continuous profiling and optimization system could effectively control the PDT.

Instead of only disabling the access to a cache way, we propose switching off the ways completely. We have developed a dynamic control for the same and combine it with the drowsy cache line scheme.

A mechanism to identify an application's I-cache requirements in order to reduce the leakage current is proposed in [13]. Using a threshold scheme, the mechanism reacts to changes in miss rate by changing the number of sets in the cache. The proposed mechanism uses a variable set *mask* to properly access the corresponding set. The optimum '*miss bound*' is searched by running simulations on a per application basis. Comparison results with this scheme are presented in Section 7.3.

CHAPTER 3
MOTIVATIONAL EXAMPLES

We studied the general behavior of embedded applications with regard to cache accesses. We used 19 applications from the MiBench/MediaBench benchmark suite and recorded the cycles between successive hits to the same cache line. Figures 1 and 2 show the data averaged over lines in a 32 KB Icache and a 32 KB Dcache, respectively. Most of the cache line reuse is concentrated within the first few hundred cycles; the number of hits after the '5000 cycles' interval is negligible. The figures show the average and maximum number of accesses in every interval. Different applications have their accesses concentrated in different bins. Also, the standard deviation shows considerable variation within a single bin. To effectively manage the drowsy lines, we need dynamic control based on runtime application behavior.

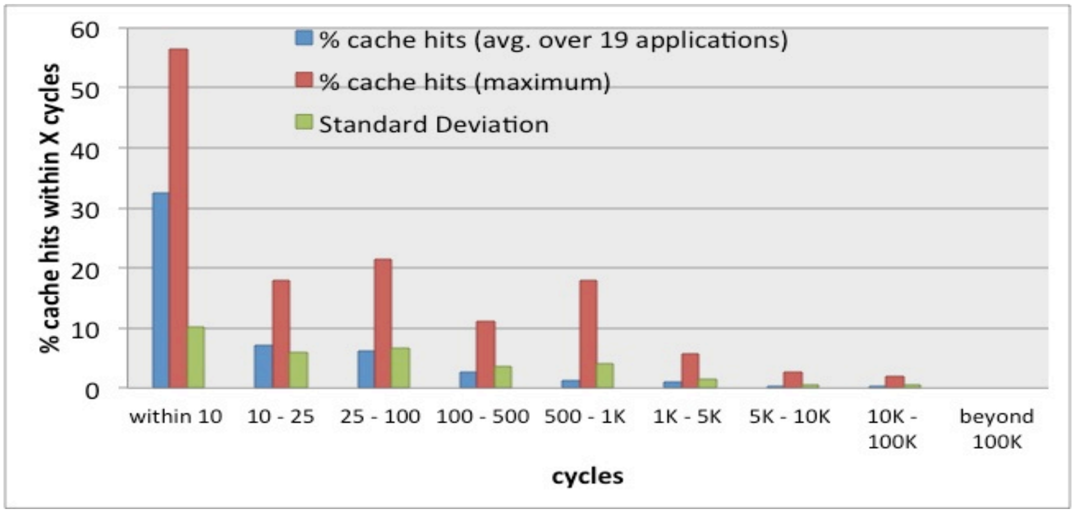

Figure 1: Access Pattern in a 32KB Icache

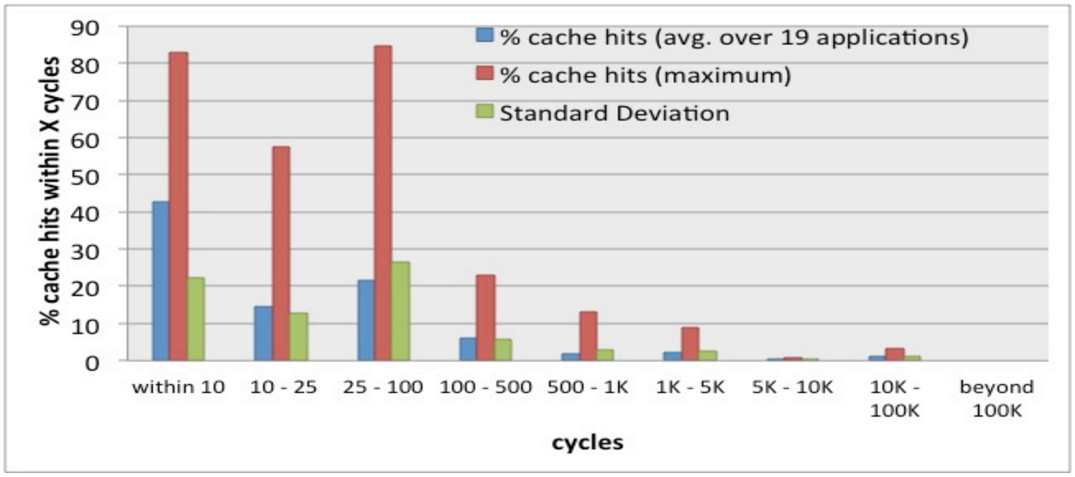

Figure 2: Access Pattern in a 32KB Dcache

CHAPTER 4
CIRCUIT IMPLEMENTATION

4.1 Additional circuit for leakage reduction in cache lines

To achieve reduction in leakage power, we assume the drowsy cache lines implementation that uses voltage scaling, as was done in [3]. This drowsy voltage is set to be approximately 1.5 times the threshold voltage Vt. A typical value for drowsy voltage for today's technology is 0.3V. Figure 3 shows the associated circuit design from [3]. We apply this drowsy mechanism only to the data array cells of the cache. The tag array is always kept awake: it contributes only about 5% of the total leakage in a 32 KB cache (Cacti 5.3) [14]. Furthermore, if tags are also put to drowsy mode, then the cache hit latency increases, since the drowsy tags have to be woken up before tag comparison can be done. Implementing this voltage scaling technique gives around 71% reduction in leakage power for individual cells. This is significant since data array cells contribute about 57% of the leakage power in a 32 KB cache (Cacti 5.3). This number increases with an increase in the size of the cache.

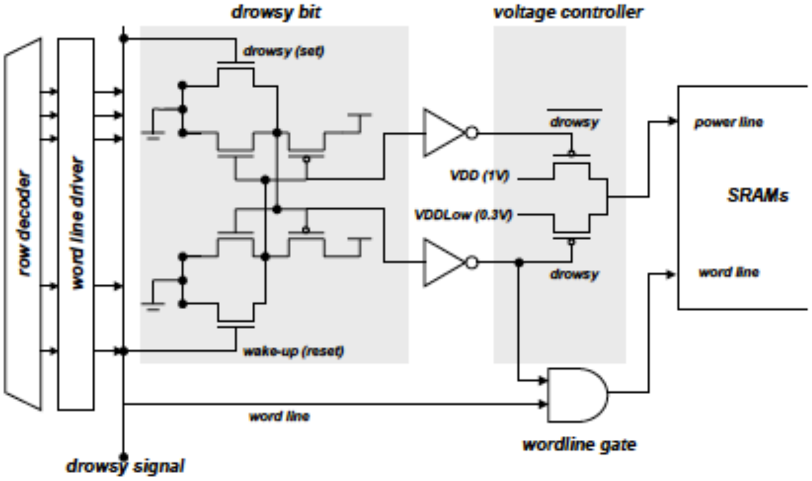

Figure 3: Implementation of drowsy cache line [3]

The transition delay between the two voltage modes for a cache line depends on the size of the pass transistors in the voltage controller. As per [3], a 64 × Leff transistor has an access time of 1 cycle, while a 16 × Leff transistor has an access time of 2 cycles. However, making the transistors wider increases the dynamic power dissipation. Since the wakeup latency is critical for the performance of the processor, we maintain it at 1 cycle, but we have selected a smaller transistor for pulling down the voltage, and conservatively selected the pull down time to be 3 cycles.

As presented in [3], this area overhead is less than 3% per cache line (taking into consideration the bigger voltage controller and the circuit for drowsy bit). The cache line size is 32 bytes.

It must be noticed that [3] considers individual voltage controller for every cache line. This means that the Vdd contacts cannot be shared for adjacent memory cells. This leads to a major area overhead in the design of the SRAM array (which has not been taken into account in the previous calculation). This overhead can be removed by having a single voltage controller of a pair of cache lines that share Vdd contacts. Section 5.4 discusses the algorithm where we control a pair of cache lines together.

4.2 Additional circuit for counters

The basic idea of our design is that a cache line that has not been accessed in the last few cycles, should be put into the drowsy state. The optimum value of this *drowsy interval* is dependent on the behavior of the application and is obtained adaptively during operation.

To implement this we need ideally a dedicated counter for every cache line, but the area and power overheads of this design would be prohibitive. Hence, we use a combination of global and local counters [6]. Every cache line has a small counter, for example, a 2-bit counter. A maximum count is set for the global counter according to the drowsy interval to provide an increment signal to the local counters. The local counter counts the number of such increment signals from the global counter. When the local counter saturates, the drowsy bit for the cache line is set, and the line is put to drowsy mode. Whenever there is an access on a cache line, the local counter for the line is reset. This method has an inherent error. The state of the global counter is independent of the accesses to the cache lines. For a particular cache line, if the local counter receives a signal just after an access, the state is incremented immediately. Hence, the count in this scheme will not be accurate.

We have observed that the 2-bit approximate counting scheme has an average energy consumption penalty of only 0.33% and a maximum penalty of 1.26% over the exact counting scheme. Furthermore, based on the behavior of the application, some applications perform better with the approximate counting scheme. We have also tried a scheme with 1-bit counter. In this scheme, there are only two counter states - active and drowsy. Hence, at every signal from the global clock, all the cache lines are put to drowsy mode. This mechanism is essentially the same as having no local counters and using just a global counter to put the entire cache to a sleep mode whenever it saturates. This scheme has an average penalty of 1.82% with a maximum penalty of 11.41%. The behavior is highly dependent on the application and the selection of drowsy intervals. Hence, we have selected the 2-bit counting scheme for our design.

The overhead of the global counter is negligible in comparison to the transistors in the entire processor. We have added a 30-transistor overhead to every cache line, for the 2-bit counter [6].

Combining the overhead of the leakage reduction circuit and that of the local counters, we get an increase of 3.9% transistors per cache line. Thus, in our simulation, we have assumed an overhead of 3.9% in leakage power. Additionally, we have added an overhead of 5% dynamic power, due to the extra routing that is required. Note that 5% is a conservative number as the circuit for the drowsy bit and voltage controllers have very low switching activity.

CHAPTER 5
DYNAMIC ALGORITHM FOR DROWSY CACHE

5.1 Algorithm for Icache

We begin with implementing our algorithm for Icache. For dynamic reconfiguration, we searched for performance counters that correspond well with the energy consumption in the processor. We observed that the Instruction Fetch Rate (IFR) is closely related to the Icache accesses on drowsy lines. When there is a hit on a drowsy line, no instructions are fetched in that cycle. As the number of hits on drowsy lines increases, the Instruction Fetch Queue (IFQ) starts running dry, which in turn affects the pipeline speed. We propose to use a counter that counts the number of instructions fetched from the Icache in every cycle. Such a counter is present in modern processors like Intel Xeon and Core i7. The average IFR is calculated by dividing this count by the number of cycles passed since the last measurement. The counter is reset after making a measurement. In our profiling, we observed that the *drowsy interval* showing best energy benefits also shows a sharp increase in the average IFR. We developed a simple algorithm that measures the IFR runtime and selects a near-optimal value of the drowsy interval. We first discuss our algorithm assuming that every cache line has a dedicated Vdd line and voltage controller. In Section 5.4, we show the extension of this scheme to the design where Vdd contacts are shared between adjacent lines.

Our algorithm relies on a learning process to determine the best drowsy interval. We start with an initial period that is set to the maximum interval considered. We then keep reducing the interval until the average IFR reduces by more than a pre-determined *threshold ratio*. To estimate the average IFR for a given value of the drowsy interval, the same interval is repeated *n* times (n is called the *repeat count* and is a parameter of the algorithm). After *n* repetitions of an interval, the average IFR is calculated. If the ratio between the current average IFR and the previously calculated IFR is smaller than the *threshold ratio*, learning is stopped. Once the learning is over, the system goes into tracking mode. In this mode, we take a measurement of the average IFR in every epoch. If the average IFR has decreased by more than the *threshold ratio*, then the learning cycle starts again. We have also tried schemes were learning is restarted if the average IFR increases above a certain threshold, but no added advantage was observed. We have studied the sensitivity to the parameters of this algorithm, i.e., *repeat count* and *threshold ratio*. The experimental results are discussed in Section 6.3.

5.2 System call overhead of Dynamic Learning

When we call the learning function, there is a context switch and the application is stalled. We have quantified the overhead due to this context switch and included it in our simulations. The context switch overhead can be classified into two parts: the direct overhead (number of cycles spent in the system call), and the indirect overhead (cache pollution due to the system call). According to [25], the system call overhead of *getpid* (the shortest Linux system call) is 223 cycles. Also for the native Linux kernel, the architectural overhead for entering and leaving kernel mode was shown to be 82 cycles. For our learning function, the major computation overhead comes from the floating-point calculations involved in computing the averages, calculating the ratios and comparing them. Taking these into account, we estimate a 300-cycle overhead for every call to the learning function.

[26] shows the size of the cache footprints for various system calls. The Icache pollution due to system calls is limited to a few 10s of cache lines [26][25]. Since our code for the learning function is relatively small, we have estimated the overhead to be 50 cache lines. In our simulation, a random selection of 50 cache lines is invalidated for every call to the learning function. No pollution for the Dcache has been considered, as our learning function need not access the data memory. We have performed a sensitivity analysis on our algorithm, with different values of system call overhead, the results of which are presented in Section 6.5.

5.3 Algorithm for Dcache

We studied the impact of implementing a similar dynamic algorithm for Dacche. No widely prevalent performance counter correlates well with the Dcache drowsy access pattern. Hence, we use a new performance counter that counts hits on drowsy Dcache lines. We calculate the percentage hits with respect to the total number of cache accesses, and use this to control our learning algorithm. This counter can be updated with the wake-up signal given to a drowsy line. A wired-OR logic implementation will be required for the same. The counter is cleared by software at the start of the learning cycle for every interval.

In our experiments, we observed that the dynamic scheme for Dcache does not provide considerable advantages over the static scheme. The advantage of the dynamic scheme is mainly observed in the Icache. This is because the processor performance is closely dependent on the behavior of the Icache. In our processor, 4 instructions are fetched in every cycle (fetch, decode, issue width is 4). When there is a hit on a drowsy line in the Icache, no instructions are fetched in that cycle. This means that the instruction fetch queue may run out of instructions and the superscalar is unable to issue instructions at its maximum potential, resulting in degradation in performance. In contrast, the effect of Dcache is not so profound. The access rate of Dcache is lower than that of the Icache. Furthermore, when there is a hit on a drowsy line in the Dcache, it adds a latency of 1 cycle to just that instruction. Based on the data dependencies, there may or may not be a latency on instructions down the line. Also, many of the hits on drowsy lines in the Dcache may be hidden by other hazards in the pipeline.

Since the static scheme works satisfactorily for the Dcache, the added design overhead for the dynamic scheme is not justified. It is for this reason that we implement our dynamic scheme only for the Icache. Section 6.4 discusses comparison results when using dynamic and static schemes for the Dcache.

5.4 Selecting Granularity - Sharing Vdd lines across adjacent cells

As discussed above, adjacent rows in SRAM arrays generally share the Vdd contacts. Controlling every row individually leads to a major area overhead. We can decrease the granularity of our control to reduce this overhead. We have a single voltage controller for the pair of lines but every line has its own 2-bit local counter. We follow a similar algorithm as discussed in the previous sections. We have tried out two flavors.

1. Either Counter Saturate (ECS) - Switch off the pair cache lines if either of the two associated counters saturate.

2. Both Counters Saturate (BCS) - Switch off the cache lines only after both the associated counters saturate. This is similar to having a single 2-bit counter for a pair of cache lines. This counter is reset on access to any of the two cache lines. When this counter saturates, it means that no access has been made to any of the two cache lines in the drowsy interval.

The former is a much more aggressive scheme and provides more energy benefits at the expense of higher performance degradation. The experimental results are presented in Section 6.2.

CHAPTER 6
EXPERIMENTAL RESULTS

6.1 Experimental Setup

We have used the Simplescalar tool [2] to estimate the performance of our scheme. Sim-Wattch1.02d [1] is used to simulate power and Cacti version 5.3 [14] is used to get the scaling factors for leakage power in Dcache and Icache. These scaling factors have been included in the Sim-Wattch code to generate results for leakage power. We ran our simulations on randomly selected 19 applications from the MiBench [16] and MediaBench [18] suite. These benchmarks contain representative applications for embedded processors. We have modeled a generic superscalar processor for our experiments. Table 1 shows the base line configuration of our processor. The processor has 32 KB, 8-way associative Icache and Dcache.

Clock rate	2Ghz
Process parameters	45 nm
Threshold voltage	0.2 V
Supply voltage	1.2 V
Fetch, issue, decode, commit width	4
Instruction fetch queue (IFQ) size	16
Load-Store queue (LSQ) size	32
ROB	64
Branch predictor	2K entry, bimodal
Integer functional units	2ALUs, 1 mult/div
FP functional units	2ALUs, 1 mult/div
L1 Icache	32KB, 8-way
L1 Dcache	32KB, 8-way, writeback
Combined L2 Cache	256KB, 4-way, writeback
L1 hit time	1 cycle
L2 hit time	20 cycles
Main memory hit time	100 cycles (first chunk), 6 cycles (inter chunk)

Table 1: Baseline configuration of the processor

6.2 Performance Results of our schemes

We have simulated the performance of both the schemes discussed in Section 5.4 - ECS and BCS. Energy measurements are done in terms of Million Instructions Per Joule (MIPJ). We have measured the energy consumption of the entire processor, rather than only for caches. This gives us a more realistic picture of the actual benefits of our scheme, as the increase in runtime affects the energy consumption of the entire processor. It should be noted that all designs use the same algorithm parameters for the dynamic scheme – *repeat count* = 5 and *threshold ratio* = 0.9. This configuration is used only for Icache. As mentioned above, the Dcache uses the static scheme. We discuss the sensitivity to the algorithm parameters in the next section. Figure 4 shows the MIPJ benefits with respect to the base processor (with no drowsy caches). Figure 5 shows the IPC (Instructions per cycle) degradation with respect to the base processor.

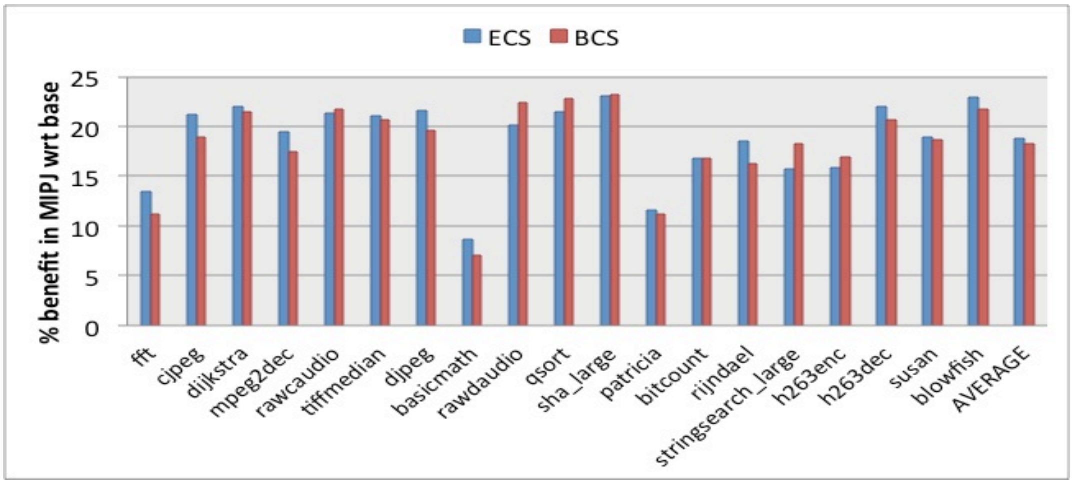

Figure 4: Benefits in MIPJ for ECS and BCS schemes

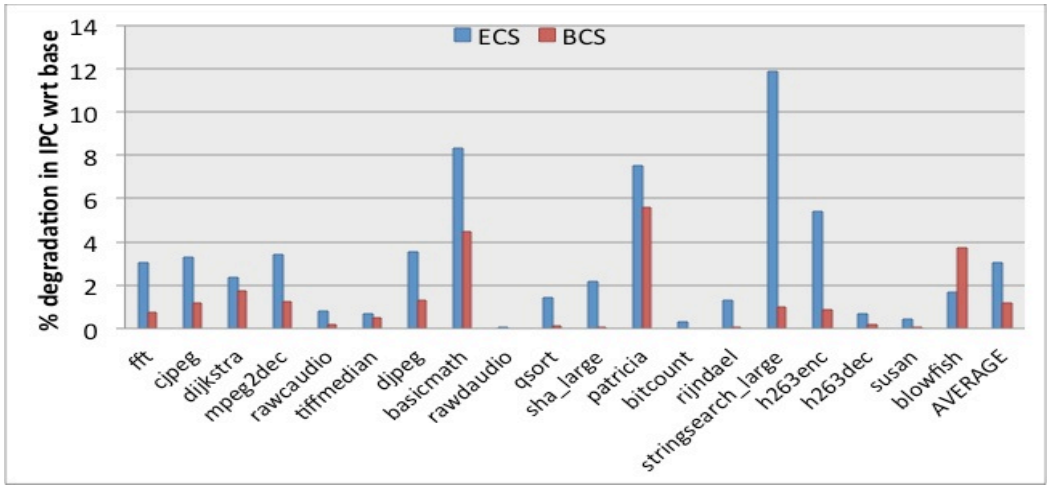

Figure 5: Degradation in IPC for ECS and BCS schemes

As seen in the figures, the MIPJ benefits achieved by ECS are only slightly higher than BCS. The ECS scheme is expected to provide higher MIPJ benefits, as it is more aggressive in putting lines to drowsy mode. However, most of the benefits are offset by the increase in runtime. The BCS scheme provides similar MIPJ benefits with much lower IPC degradation, as it tracks cache accesses more accurately.

6.3 Sensitivity to algorithm parameters

In order to find the best set of parameters for our dynamic algorithm, we studied the sensitivity of our algorithm to different parameter values. The maximum drowsy interval selected in our design is 5000 cycles. This decision is based on our observation in Figures 1 and 2, that very few cache hits occur after that interval. The epoch mentioned in the tracking mode is set at 100,000 cycles. We vary one parameter at a time and measure the average benefit in MIPJ with respect to the base processor. We individually check the effect of varying the IFR *threshold ratio* and the *repeat count*.

Figure 6 shows the behavior of a 32 KB cache with different values of the repeat count. The IFR threshold ratio is maintained at 0.9. It is seen that for very low and very high periods, the benefits decrease slightly. When repeat count is high, then the time taken to find the optimal drowsy interval is also high and the applications spend a lot of time running on sub-optimal configurations, which results in lower benefits. Also, using *repeat count* values of 1 or 2 leads to unreliable learning. As we have discussed, the state of the counters are independent of the time when a new *drowsy interval* is initiated. For example, a 2-bit local counter might already have counted half way through when a new drowsy interval is introduced. Hence, the application needs to 'settle down' to the *drowsy interval* being tested. A value of *repeat count* between 3-5 works best. We choose the value of 5 for all our comparisons.

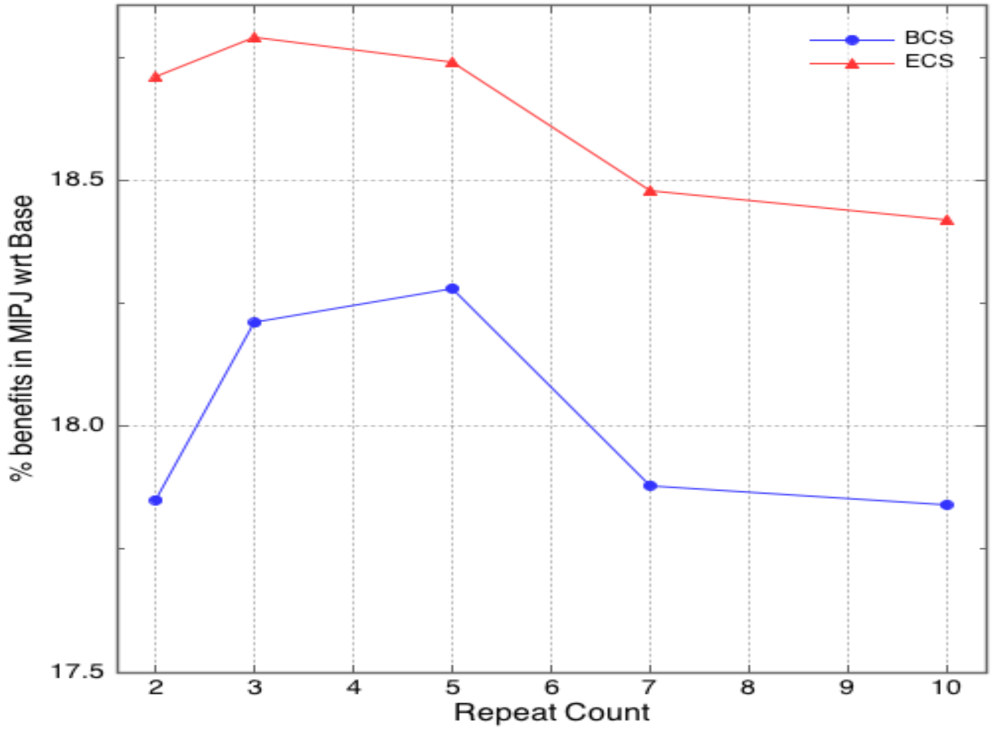

Figure 6: Sensitivity to *Repeat Count*

Figure 7 shows the effect of varying the threshold ratio. For this case, we have maintained the repeat count at 5. From the graph we see that the optimal value of the *threshold ratio* is 0.8 - 0.9. We select the value of 0.9 as it works slightly better. As we decrease the *threshold ratio* to smaller values, very short *drowsy intervals* are selected. This means that the lines are put to drowsy mode at very short intervals, leading to an increase in the performance degradation. With decreasing values of *threshold ratio*, the ECS scheme degrades very quickly. This is because of the aggressive nature of the ECS scheme. Here, two lines are switched off even if one of the counters saturates. Hence, in this mode, the number of lines in drowsy mode increases very quickly with decreasing *threshold ratio*.

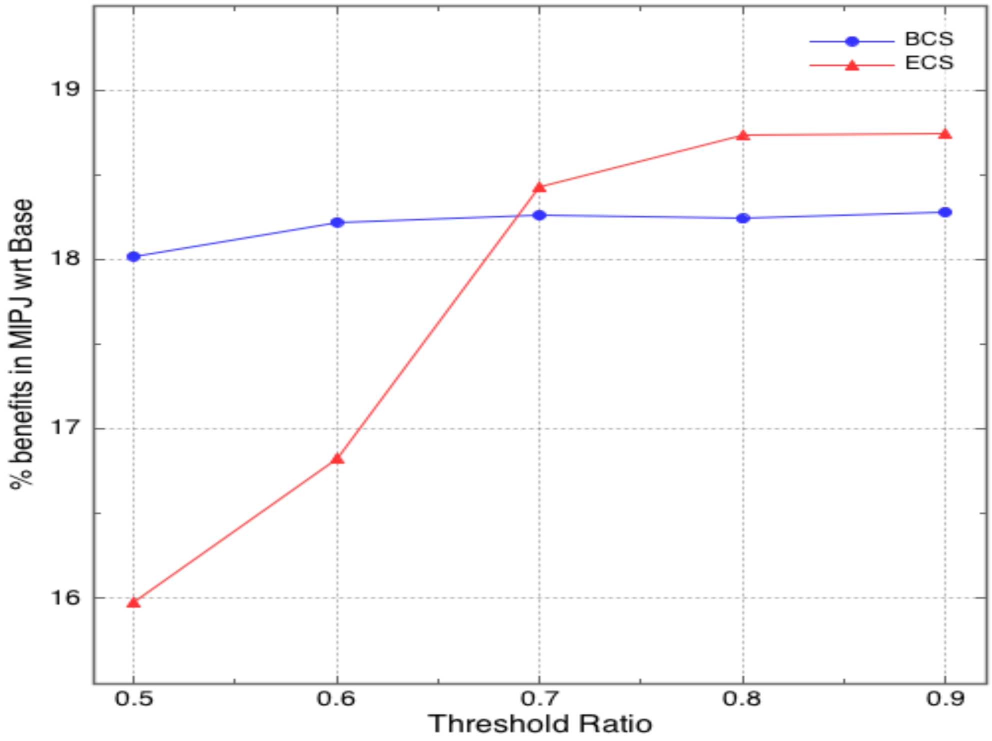

Figure 7: Sensitivity to *Threshold Ratio*

For our final selection we use IFR *threshold ratio* = 0.9 and *repeat count* = 5. We have verified these parameters with cache sizes of 16KB, 8KB and 4KB and observed a similar trend. The same set of parameters works satisfactorily for different cache sizes. We discuss this in Section 6.6.1.

6.4 Configuration for Dcache

We have analyzed the static intervals that work best for the Dcache. The intervals range from 100 to 5000. For the ECS scheme, the best *drowsy interval* is 5000 while for the BCS scheme the best *drowsy interval* is 100. The ECS scheme puts two lines to drowsy mode based on the access to any one of the lines. The other line might have a different access behavior. Hence, this requires a conservative selection for the *drowsy interval* to balance out the effect. On the other hand, the BCS scheme by nature is conservative and hence an aggressive selection of the *drowsy interval* works best with it. It must, however, be noted that, the selection of these intervals for Dcache does not have a considerable impact. When comparing the benefits in terms of MIPJ between a drowsy interval of 100 and 5000, we see that the average difference is around 0.6% while the maximum difference is approximately 3.7% for *Qsort*.

We compared the MIPJ benefits between static and dynamic schemes for Dcache. The dynamic scheme for Dcache uses the same algorithm parameters as that for the Icache. Both static and dynamic schemes show very similar results. The average difference is only 0.49% with the maximum difference of 2.7% for *Rijndael*. The differences in IPC results are smaller with only three applications showing differences more than 1%. It is for this reason that we have chosen a static scheme for Dcache.

6.5 Sensitivity to System Call Overhead

As discussed in Section 5.2, we have included a direct overhead of system calls, of 300 cycles for every call to the learning function. We have also included a cache pollution of overhead of 50 cache lines every time the function is called. In this section we show the sensitivity of our algorithm to variation in these system call overheads. Figure 8 shows the sensitivity to the direct overhead of system call, in terms MIPJ benefits. Even for an overhead of 1000 cycles the degradation is less than 0.5% for both the schemes. The minor decrease in the benefits is due to the increase in runtime of the applications. Figure 9 shows the sensitivity to cache pollution in terms of MIPJ benefits. For this experiment, the system call overhead is maintained at 300 cycles. For a cache pollution of 100 lines, the degradation is around 0.4% for the ECS scheme and around 0.1% for the BCS scheme. Cache pollution brings in some unpredictability to the process. This affects applications if they have a large active footprint. If a large number of lines are polluted whenever the learning function is called, the system needs time to reload the lines from the memory. Note that we use a repeat count of 5, so when we try out lower drowsy intervals while learning, the configuration runs for lesser number of cycles, which gives the system less time to recover. In this case, the average IFR is affected not only by hits on the drowsy lines but also cache misses. Hence, for lower drowsy intervals, the average IFR will tend to be lower than it ideally should. Thus, when the cache pollution is high, there is a smaller chance that these lower intervals are selected. This might lead to a sub-optimal selection. Figures 8 and 9 show that the effect of cache pollution is much more than that of stalls due to direct overhead of system calls.

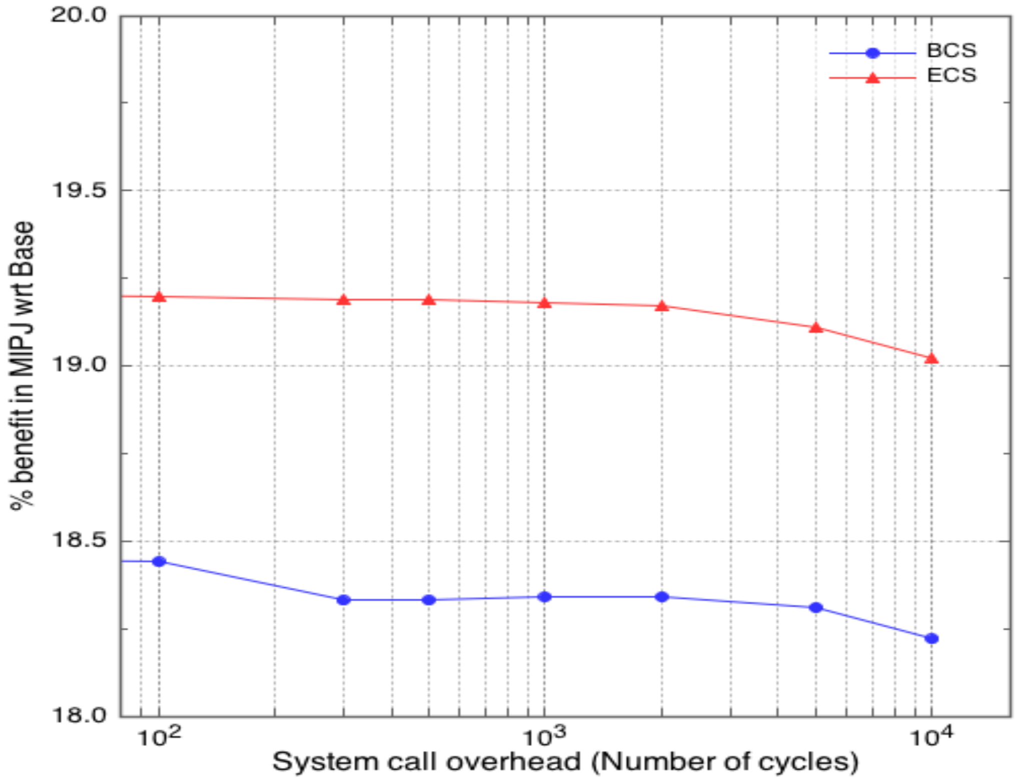

Figure 8: Sensitivity to Direct overhead of system calls

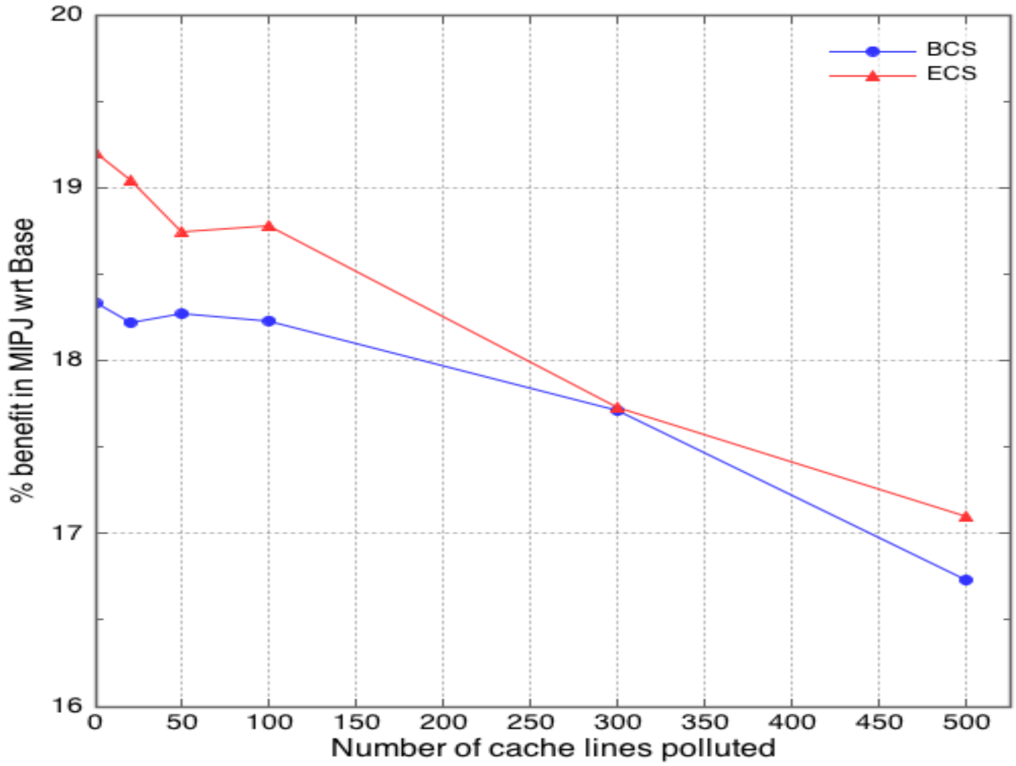

Figure 9: Sensitivity to cache pollution overhead

6.6 Our dynamic scheme vs. Earlier Methods

As we have seen in Figures 3 and 4, the BCS scheme show satisfactory results with lower performance degradation (when compared to ECS). Hence, we choose BCS for the rest of the experiments.

6.6.1 Comparison with Static scheme

In this section we compare our dynamic scheme to the static schemes presented in [3]. We use the *NoAccess Policy* from [3] which shows better results than the *Simple Policy* [3], for the static scheme. The *NoAccess Policy* puts only those lines to drowsy mode that are not accessed within the *drowsy interval*. The *Simple Policy* puts all lines to drowsy mode when the *drowsy interval* elapses. For a fair comparison, we have modified the design in [3] so that it uses shared Vdd contacts for adjacent cache lines. We use the BCS design for the static scheme as well. Here, the selection of drowsy intervals does not change in runtime. Although [3] does not discuss the implementation details of the *NoAccess Policy*, we have used the same approximate counting scheme involving global and local counters. We have calculated the percentage benefits in terms of MIPJ with respect to the base processor (with no drowsy lines). The best static scheme for both Icache and Dcache is selected on the basis of MIPJ benefits by profiling. Figures 10 and 11 show the comparison for MIPJ benefits and IPC degradation, respectively. We see that the MIPJ benefits of both the schemes are similar. But the dynamic scheme shows considerably lower performance degradation with respect to the static scheme. The maximum difference in the performance degradation is around 5% for *Stringsearch* and the average difference between the two schemes, for the 19 applications, is around 1.5%. It can be clearly seen from the figure that many applications show a big improvement with the dynamic scheme. These results can be explained by the fact that we have chosen the configuration of the static scheme on the basis of the best MIPJ benefits. In the event that the static configuration is chosen to minimize the performance degradation, the energy benefits come down considerably.

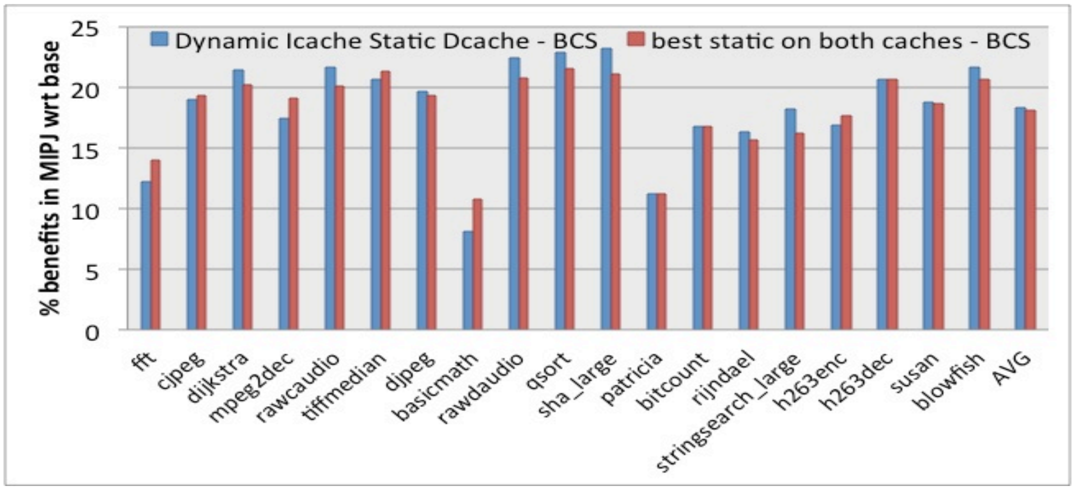

Figure 10: Comparison of MIPJ benefits (static and dynamic schemes)

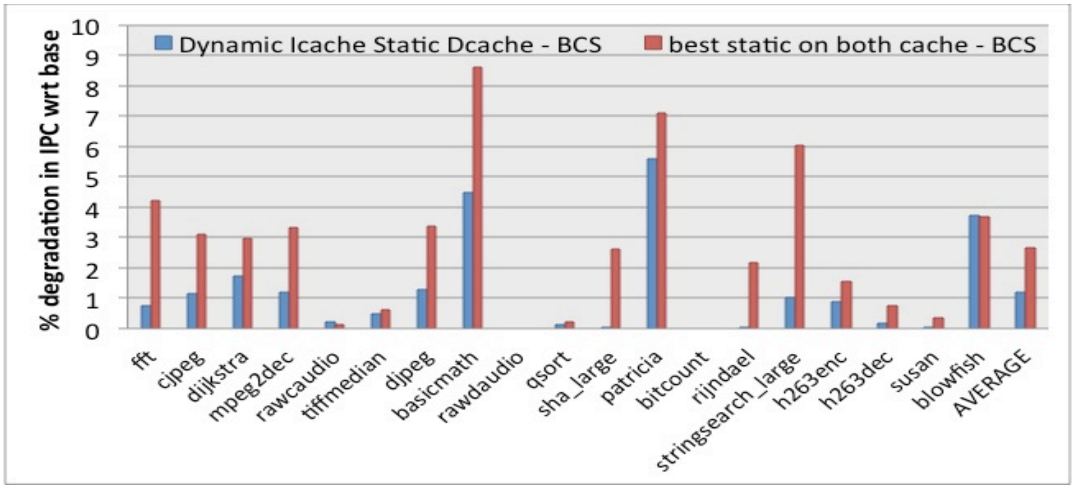

Figure 11: Comparison of IPC degradation (static and dynamic schemes)

We have also studied application behavior across cache sizes. Applications can show widely varying behavior for different cache sizes. Figures 12 and 13 show two kinds of behavior. For *Mpeg2dec*, the optimal interval remains at 500 across all cache sizes. In contrast, the optimal interval for *Rijndael* varies. The energy minima occur at higher intervals for cache sizes of 32 and 16 KB. But for cache sizes of 8 KB and 4 KB, the energy minima are at the lowest interval of 100. This is because, for cache sizes of 8 KB and 4 KB the entire cache starts trashing. Due to capacity misses all the lines are trashed before being reused. Hence, there is no point in keeping the lines awake till 5000 cycles. The best energy benefits are seen when the lines are put to sleep after 100 cycles (the lowest configuration used in the design). For another application, *Basicmath*, we observe that the optimal drowsy interval increases from 500 to 2000 when the cache size reduces from 32 KB to 4 KB. This happens because, in this case, only a part of the active code footprint is trashed, while the rest is always present in the cache. Because of these extra misses, the cache hit distance (the number of cycles after which a line is reused) increases for the lines that are always present in the cache. Therefore, instead of 500, the optimal drowsy interval increases to 2000.

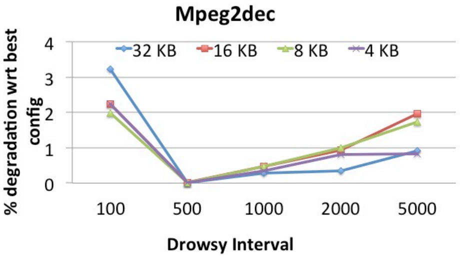

Figure 12: Dependence of optimal interval on cache size: *Mpeg2dec*

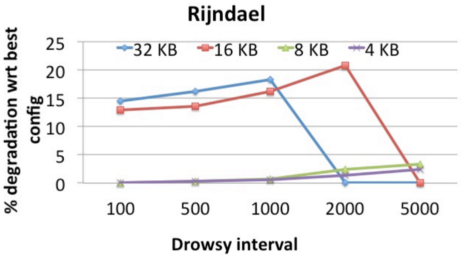

Figure 13: Dependence of optimal interval on cache size: *Rijndael*

To demonstrate the robustness of the dynamic scheme, we show results for *Rijndael*. We used our dynamic algorithm with the same parameter values as used for the 32 KB cache. Figure 14 shows that the dynamic algorithm is able to find the best configuration across cache sizes.

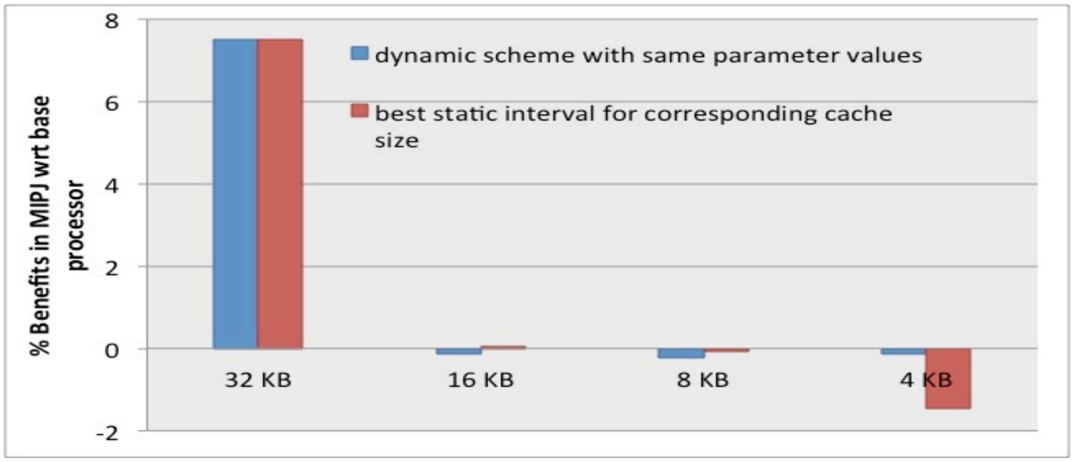

Figure 14: Effectiveness of dynamic algorithm across cache sizes: *Rijndael*

6.6.2 Comparison with Modified MRU (Most Recently Used) Scheme [32]

We have implemented the dynamic scheme presented by Zushi *et. al,* [32]. As mentioned in Chapter 2, this scheme is an improvement on the MRU scheme proposed by Petit *et. al,* [11]. Along with the MRU information, the scheme in [32] also uses a time window to control the voltage mode transition of the cache lines. There is an access bit associated with every cache line. This bit is set whenever an access is made to the cache line. At the end of every window, a global drowsy signal is sent to all cache lines. Based on the MRU information and the access bit, a decision is made for every cache line whether to put it to drowsy mode. The access bits are then cleared for the next window. The policy only limits the number of awake lines in the beginning of the window. It does not limit the number of lines that can be kept awake during the window. This means that any line accessed during the window is kept awake till the end of the present window. Two different schemes are proposed – AOM (Accessed Or MRU) and AAM (Accessed And MRU). In the AOM scheme, all lines are put to drowsy mode except the MRU lines, and the lines that have been accessed in the previous window. In the AAM scheme, all lines are put to drowsy mode except those MRU lines that have been accessed in the last window. If the MRU line is not accessed in the last window, it is put to drowsy mode.

The design presented in [32], like the others, assumes a dedicated Vdd for every cache line. For the sake of comparison, we have modified the design so that these Vdd contacts are shared amongst two adjacent lines. This has some direct implications on the algorithm. Based on the policy (AOM or AAM), if a line qualifies to be kept awake in one set, then it will have to be kept awake in the adjacent set also, even if the line in the adjacent set does not qualify to be kept awake. As presented in [32], we have used a window size of 4096 cycles. Figure 15 compares the MIPJ of our dynamic scheme (BCS) with that of the AOM and AAM schemes. Figure 16 shows the comparison of the IPC degradation.

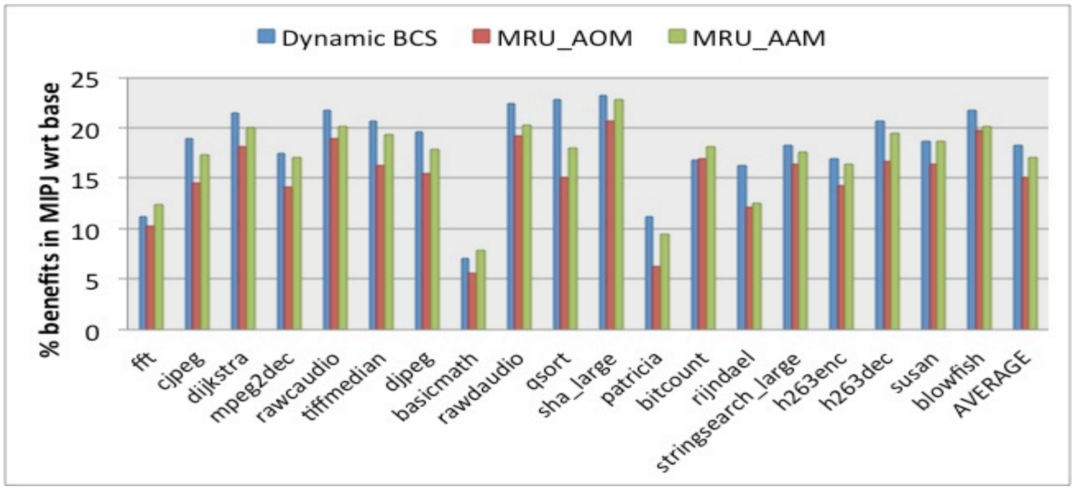

Figure 15: MIPJ benefits (Dynamic BCS and Modified MRU scheme)

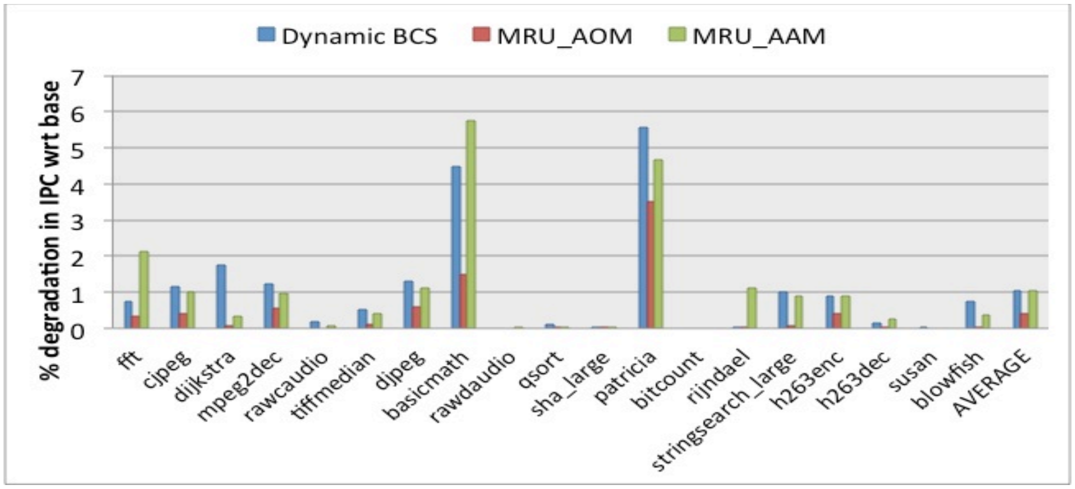

Figure 16: IPC degradation (Dynamic BCS and Modified MRU scheme)

The dynamic BCS scheme clearly shows bigger energy benefits for all applications with respect to the AOM scheme. The maximum difference in benefits is around 8% for *Qsort*, and the average difference is more than 3%. The average performance degradation of the AOM scheme is better by around 0.6% when compared to the BCS scheme. The energy benefits of the BCS scheme are slightly better than the AAM scheme, with the average difference of around 1%. The maximum difference in benefits is around 4% for *Rijndael* and 4.8% for *Qsort*. The average performance degradation is almost the same for the BCS and the AAM scheme. Some applications like *Basicmath*, *Fft* and *Rijndael* show higher performance degradation with the AAM scheme. These applications have a big active footprint in the cache, but the AAM scheme keeps awake a *maximum* of one line awake in every set. This leads to an increase in the performance penalty. Another disadvantage of the MRU-based schemes is that they cannot be used for fully associative or direct mapped caches. The benefits are also dependent on the associativity of the cache. This is more evident in the AOM scheme. For example, in a 2-way associative cache, *at least* half of the cache lines will always be awake when using the AOM scheme. Hence, the energy benefits reduce considerably. For a 32 KB cache, 2-way associative, the MIPJ benefits of the AOM scheme comes down to 6.71%. The BCS scheme provides MIPJ benefits of 13.41% for a 2-way associative cache. Likewise, the AAM scheme, when implemented in a direct-mapped cache, acts like static scheme with a window size of 4096 (since all lines are MRU in a direct-mapped cache). The AOM and AAM schemes also require a global routing for the drowsy update signal, and additional gating logic to block the drowsy signal when the MRU and/or the Access bits are set for a cache line.

6.6.3 Comparison with the Improved Drowsy Scheme [31]

As mentioned in Section 2, Alioto *et al.* [31] have proposed a scheme where they put active lines to drowsy mode immediately after the access moves on to another line. The authors have named it the *Improved Drowsy* scheme, which is an improvement over the scheme proposed by Flautner *et al.* [3]. In the *simple policy* proposed by Flautner *et al.*, all the lines are put to drowsy mode after every *drowsy window* (selected to be 4096 cycles). To implement this, all cache lines are controlled by a global drowsy update signal.

In the proposed *Improved Drowsy* scheme [31], along with this global update signal, every cache line has four other drowsy signals. The cache line gets drowsy signals from two lines above and below it. In other words, every line sends drowsy signals to two lines above and below it. Whenever a line is awake, it activates the drowsy signal on these nearby four lines. They are put to drowsy mode if they are not being currently accessed. The global drowsy signal is still activated after every *drowsy window*. This scheme works efficiently for sequential accesses but degrades quickly if the number of branches increases. Figure 17 compares the MIPJ of our BCS scheme to that of the *Improved Drowsy* scheme. This comparison has been made for the 8KB direct mapped cache configuration used in [31]. Also, since we use the shared Vdd mechanism, we implement the *Improved Drowsy* scheme by controlling one pair of lines above and below any line.

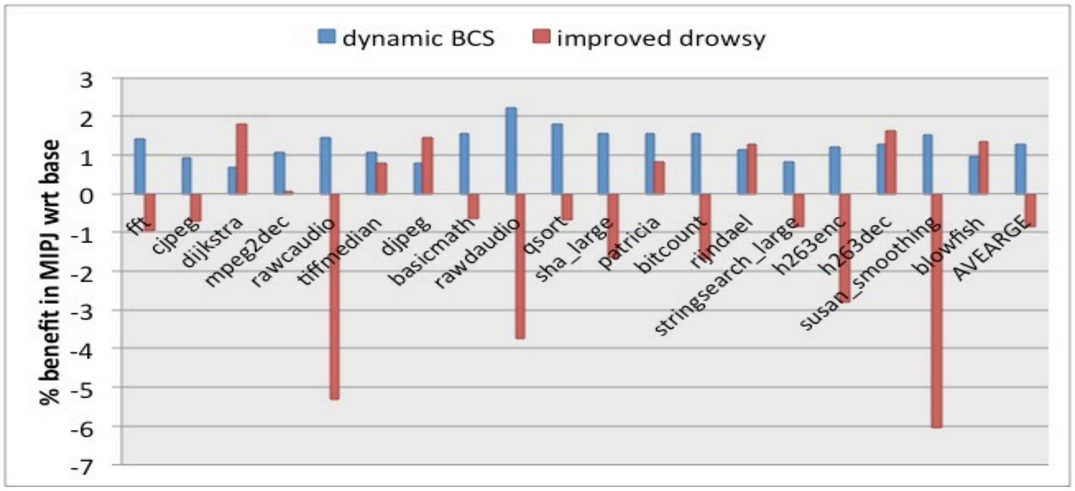

Figure 17: MIPJ benefits (dynamic BCS and *Improved Drowsy*)

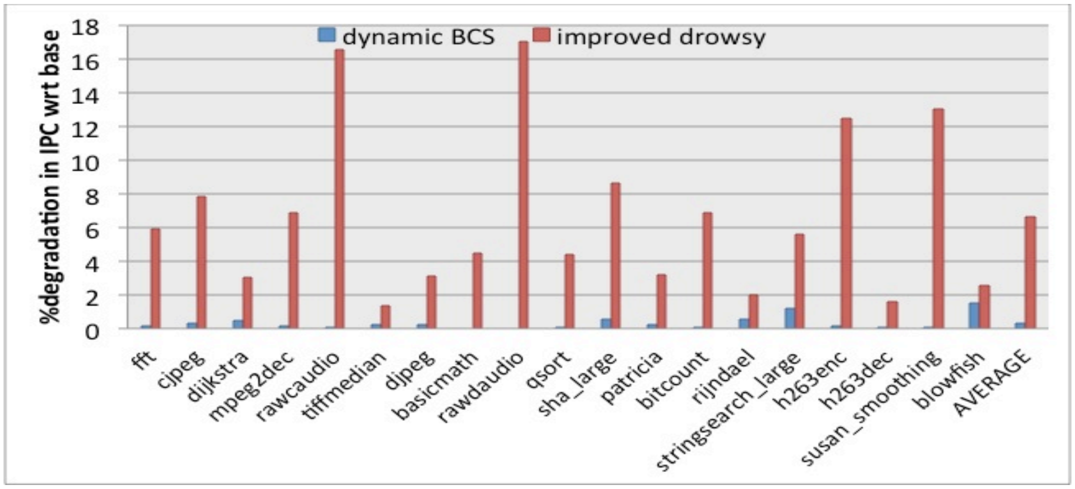

Figure 18: IPC degradation (dynamic BCS and *Improved Drowsy*)

As our simulations measure total energy consumption of the entire processor, we see that the MIPJ benefits are only around 1-2% for a 8KB cache. Figure 17 shows a comparison between the MIPJ benefits with the BCS and the *Improved Drowsy* schemes. The results show a mix of applications that perform better and worse with the Improved Drowsy Scheme. Some applications show a considerable degradation in MIPJ. These applications have a high percentage of branch instructions, for example 31% for *Rawcaudio*, 20% for *Bitcount* and 15% for *Fft*. On the other hand, applications that show higher benefits with the *Improved Drowsy* scheme have a low percentage of branch instructions like *Djpeg* (7%), *H263dec* (9%) and *Rijndael* (6%). This shows that the scheme is highly dependent on the locality of the cache accesses. Our set of applications is similar to that used in [31]. Some applications are used only in either of the two. The applications in [31] that have not been used in our simulations are *Gsm*, *Pgp*, *Rsynth*, *Typeset*, *Mad*, *Ispell*. As presented in [31], none of these applications show the highest reduction in leakage. The benefits of most of these applications are similar or lower to the average case. We do not expect the comparison results to vary, even if these applications were included in our simulations. Figure 18 shows the IPC comparison. It can be seen that the performance degradation with the *Improved Drowsy* scheme is much worse than that of the dynamic BCS scheme.

We observed that, for some applications like *Fft* and *Stringsearch*, the results presented in [31] do not match our simulations. The IPC degradation presented in [31] for these applications are much lower than shown here. This can be attributed to the different kinds of processors that are modeled in the simulator. We have modeled a 4-wide superscalar with an out-of-order pipeline implemented using Tomasulo’s algorithm. On the other hand, the authors in [31] have used an Intel XScale processor [30]. This processor issues one instruction at a time and uses three different pipelines after the RF (Register File) stage – the main execution pipeline, one for memory operations, and another for MAC instructions [29]. Instructions are allowed to complete out-of-order. The dependencies are handled using Scoreboarding [29]. Since, our processor has an issue width of 4, the penalty due to hitting a drowsy cache line is higher. Furthermore, the XScale processor uses Scoreboarding, which stalls for both RAW and WAW hazards at the Register File Read stage [29]. Hence, there is a greater chance that the penalty due to hitting a drowsy line may get hidden, as the pipeline would already be stalling on a data or a structural hazard. A pipeline based on Tomasulo's algorithm does not stall on WAW or WAR. Since, a faster pipeline incurs more penalties if instruction fetch is slow, the IPC degradation is higher for the *Improved Drowsy* scheme, for our processor.

6.7 Trend across Technology Nodes

All previous results have been generated using 45nm technology. In this section, we compare to 32 and 22nm technology nodes. Figure 19 shows a comparison of MIPJ benefits for these technology nodes. We see that there is a small reduction in the MIPJ benefits with decreasing feature size. This is because, only Vdd has been scaled down. According to ITRS 2011, the threshold voltages for high performance transistors have not been scaled down. These threshold voltages have been determined by taking into consideration, the constraint that the sub-threshold current should not exceed 100nA/μm. Due to the reduction in Vdd, the scope for voltage scaling reduces by some amount. But, we are still able to get meaningful benefits even for 22nm technology. Since, no changes have been considered to the design, the IPC results remain the same.

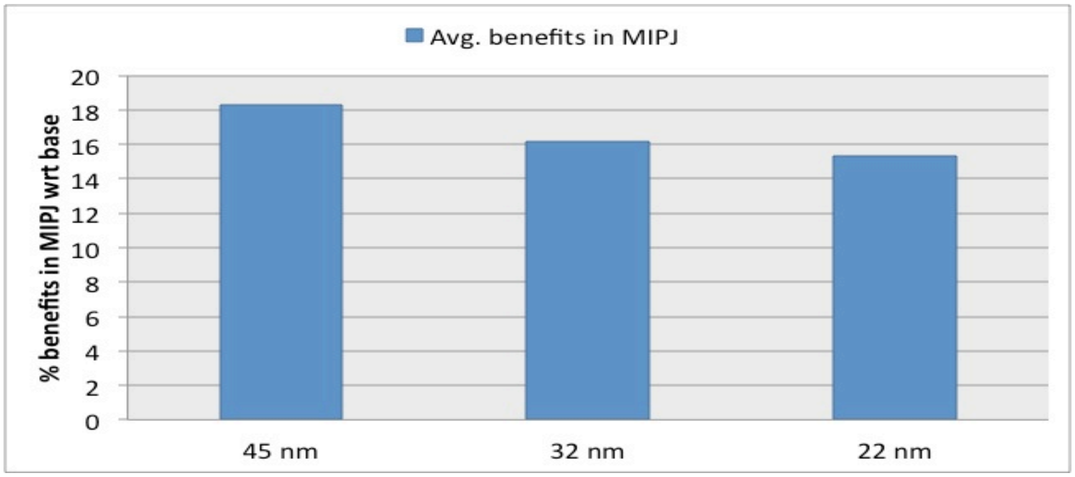

Figure 19: Trend across technology nodes

CHAPTER 7
COMBINING DROWSY CACHE LINES AND GATED-VDD SCHEMES

As discussed before, the drowsy cache scheme scales down the voltage only to a point where the data in the memory cell is not lost. This helps in preventing additional cache misses when the line is woken up. On the other hand, the Gated-Vdd mechanism completely turns off the cache lines. This results in the data being lost but the leakage energy savings are higher than that for the drowsy cache scheme. We have investigated a scheme where the combination of both can be used to maximize the energy benefits by completely shutting down some of the cache lines that are already in the drowsy mode.

7.1 Design considerations

We propose to combine the circuit for drowsy caches [3], and that for the Gated-Vdd scheme [12]. The Gated-Vdd scheme introduces a high-Vt transistor between the power rail and the SRAM cell. This transistor is switched on, when the circuit is active. To power down, this gating transistor is switched off, hence cutting the power supply from the SRAM cell. Since this is a high-Vt transistor, the leakage through this is minimal. But, using a high-Vt transistor increases the access time. Based on the exact design used, a tradeoff can be made between the access time, energy consumption and area overhead. For the purpose of our simulations, we have assumed an Nmos Gated-Vdd, dual-Vt transistor, which can provide 97% leakage benefits without any increase in access time [12]. The area overhead is 5%, which is larger than the other designs [12].

To put the cache lines in drowsy mode, we can scale down the voltage in the power line while still keeping the gating transistor in the ON state. To switch-off the lines completely, the gating transistor should be switched-off. When the line is woken up, the power line needs to be brought back to the normal mode and the gating transistor needs to be switched on.

7.2 Control Mechanism

As discussed earlier, cache lines are put to drowsy mode if they have not been accessed in the period decided by the *drowsy interval*. We can further switch them off completely if these drowsy cache lines are not woken up within a certain period after being put to drowsy mode. Like the drowsy cache scheme, these intervals are decided dynamically for Icache, and a static interval is used for Dcache.

For Icache, we use the same *drowsy interval* given by the learning algorithm. After a cache line is put to drowsy mode, we switch it off completely, if the line is not accessed for another *drowsy interval*. This can be easily implemented by extending the 2-bit local counter to a 3-bit counter. The line is put to drowsy mode when the counter reaches count 3, and is switched off when the count reaches 7.

It is not feasible to use the same design for Dcache. This is because we have used a static drowsy interval of only 100 for the BCS scheme. But, switching off the lines after another 100 cycles considerably increases the performance degradation due to additional cache misses. The static interval selected for Dcache is the same as that used by Kaxiras *et al.* for their Cache Decay scheme [6]. Their theoretical analysis suggests an optimal interval of roughly 10,000 cycles. Based on the experimental results, they propose an interval of 8000 cycles for best energy benefits. We have used the same interval for switching off the Dcache lines. We use a different technique to implement this. Since the global counter for the Dcache provides increment signals corresponding to the *drowsy interval* of 100, a very large local counter would be required to count 8000. This overhead is prohibitive. Hence we introduce another global counter that provides signals to count 8000. After the line has been put to drowsy mode, we reuse the same local counter for counting these 8000 cycles. In hardware, the local counter would use the increment signals from the first global counter when the drowsy bit is 0 (line is active). When the line is put to drowsy mode, the local counter is reset and the drowsy bit becomes 1. As long as the drowsy bit remains 1, the local counters use increment signals from the other global counter.

7.3 Experimental Results

We have implemented the proposed scheme with the parameters discussed above. Figure 20 compares the MIPJ benefits, when only using the drowsy cache scheme (dynamic BCS), with that when using a combination of drowsy cache and Gated-Vdd schemes. The results show a mix of applications that benefit from the scheme and those who do not. The benefits are not very high. This is because the added advantage of Gated-Vdd scheme with respect to leakage benefits is not very high. The drowsy voltage scheme provides around 78% leakage reduction while the Gated-Vdd scheme provides 97% (for individual SRAM cells). But, as discussed before, these schemes have only been implemented on the data array of the cache. Furthermore, the Gated-Vdd scheme results in cache misses, which increases the runtime of the application. If the increase in cache misses is too high, like that in *Patricia*, the energy consumption increases considerably. Based on the average behavior, we believe that the added design overhead of combining these schemes is not justified.

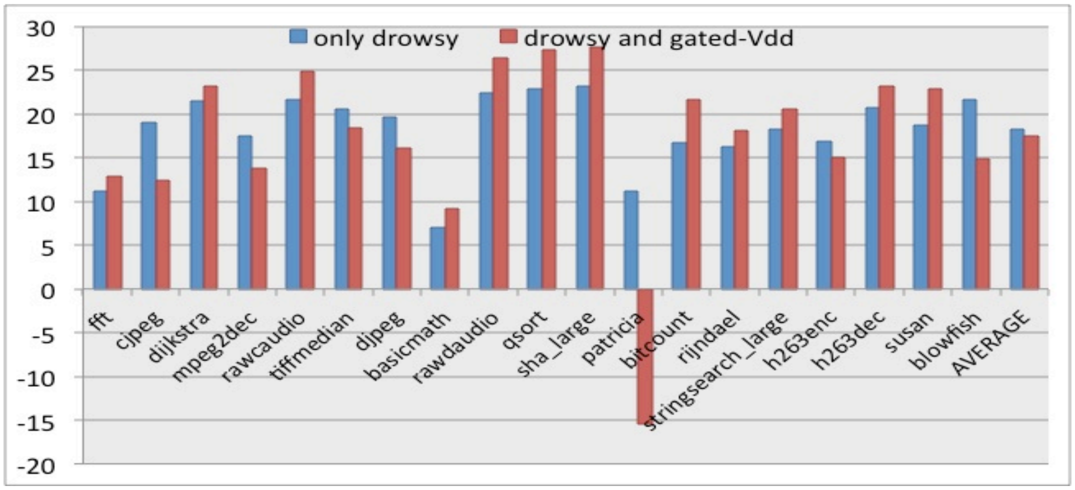

Figure 20: MIPJ benefits (*only drowsy* with combination of *drowsy and GatedVdd*)

CHAPTER 8
COMBINING DROWSY CACHE LINES AND CACHE-WAY SHUTDOWN SCHEMES

Cache-way reconfiguration is a scheme were access to entire cache ways are disabled. This is a very coarse grained mechanism but has potential to provide larger energy benefits. On the other hand, the drowsy cache scheme provides a fine-grained control over individual pair of cache lines. We propose a combination of these to schemes to gain the maximum energy benefits for a given application.

Albonesi [17] has proposed a way-selection scheme for saving dynamic power. We further switch-off the cache ways to which access is disabled, using the gated-Vdd mechanism [12], thereby saving leakage power also. This mechanism should be confused with the one presented in the previous chapter. In this scheme, the voltage to entire cache ways is gated. The drowsy cache mechanism is used on the cache ways that are not gated.

Since we are only operating on cache-ways, the address decoding remains unaltered for all the configurations. Data coherency in Dcache becomes an issue in Albonesi's design [17], as only the access to the cache-way is gated, but the data is still present in it. This creates aliasing and needs to be handled carefully. However, in our case, when switching off the cache ways, the data stored is lost. Hence, no additional mechanism is required to maintain data coherency in the Dcache. However, since we are shutting down the ways, the data that is dirty needs to be written back to the memory. We have modeled a Write Back Policy and used a Write Buffer of size 8. If the number of Dirty words is more than that, the pipeline is stalled appropriately.

8.1 Algorithm for cache-way shutdown

The structure of the algorithm is similar to that used for drowsy cache lines. We use miss rates of the cache to control the cache ways. All the cache ways are active at the start. We make a choice for the *learning interval*, which is the number of cycles for which every configuration is tried before moving on to the next one. We consecutively switch-off half the remaining cache ways, after the passage of every *learning interval*. This is done till the *threshold ratio* on the miss rates is crossed.

8.2 Integrating cache-way shutdown and drowsy cache line algorithms

We start our process with the cache-way shutdown policy. The algorithm runs independently for Icache and Dcache. Only after the cache ways have been set for a cache, is the learning for drowsy lines started for that cache. We use two different counters in this design. As mentioned before, the cache ways are controlled using miss rates while the drowsy lines in the Icache are controlled using the average IFR.

Learning for deciding on the cache ways is restarted based on the tracking of the miss rates. It should be noted that a new configuration in the cache way triggers the relearning for the drowsy lines scheme, but the reverse is not true. However, even for the same configuration of the cache ways, learning can be restarted for drowsy lines based on the tracking of the respective counters.

8.3 Experimental results

We see a considerable increase in the energy benefits. The average MIPJ benefits with this scheme is 31% and the average performance degradation is around 3%.

We have made a comparison of this scheme to the predictive MRU scheme [22], discussed in Section 2.2. The predictive MRU scheme is able to provide around 9% MIPJ benefits. We have also compared our scheme to the DRI-Icache [12][13], discussed in Section 2.2. This scheme has only been implemented for the Icache. Implementing it in the Dcache needs added control to resolve aliasing. Hence, we make the comparison only for Icache. The DRI-Icache uses miss-bound to dynamically determine the number of sets. We have implemted DRI-

Icache with various values of miss-bound. The best MIPJ benefits are 11% with a miss bound of 5000. Our scheme, on the other hand, gives MIPJ benefits of around 19% (on only Icache). We also outperform the DRI-Icache scheme with respect to IPC degradation, for all applications.

CHAPTER 9
CONCLUSION

Large memory structures, such as caches, offer the possibility of considerable reductions in leakage energy. Due to the individual characteristics and complexity of applications, a dynamic scheme is necessary for adaptive, fine-grained control over the drowsy cache lines. We have proposed a low cost, robust dynamic scheme that works across various configurations. Our scheme works satisfactorily across applications and with caches of different associativity and sizes. Compared to earlier published schemes, our dynamic scheme provides more energy benefits with lower performance degradation. For 45nm technology, the MIPJ benefits are 18.27% with IPC degradation of 1.2%. The scheme shows promising results for more recent and future technology nodes.

BIBLIOGRAPHY

[1] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architecture-level power analysis and optimizations," Proc. ISCA, 2000, pp. 83-94.

[2] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, Computer Sciences Department, Univ. Wisconsin–Madison, June 1997.

[3] K. Flautner, N. S. Kim, S. Martin, D. Blaauw and T. Mudge, "Drowsy Caches: simple techniques for reducing leakage power," Proc. ISCA 2002, pp. 148-157.

[4] M. J. Geiger, S. A. McKee and G. S. Tyson, "Drowsy Region-Based Caches: minimizing both dynamic and static power dissipation," Proc. ACM 2nd conference on Computing Frontiers, May 2005, pp. 378-384.

[5] H. Goto, S. Nakamura and K. Iwasaki, "Experimental fault analysis of 1MB SRAM chips," Proc. VTS 1997, pp. 31–36.

[6] S. Kaxiras, H. Zhigang and M. Martonosi, "Cache decay: exploiting generational behavior to reduce cache leakage power," ISCA 2001, pp. 240-251.

[7] N. S. Kim, K. Flautner, D. Blaauw and T. Mudge, "Drowsy instruction caches: leakage power reduction using dynamic voltage scaling and cache sub-bank prediction," Proc. IEEE MICRO 35, Nov. 2002, pp. 219-230.

[8] N. S. Kim, et al., "Leakage current: Moore's law meets static power," IEEE Computer, Vol. 36, Dec. 2003, pp. 68-75.

[9] A.C. Nacul and T. Givargis, "Dynamic Voltage and Cache Reconfiguration for Low Power," Conference on Design, Automation and Test in Europe (DATE), Feb. 2004, pp. 1376-1377.

[10] K. Nii, et al., "A low power SRAM using auto-backgate-controlled MT-CMOS," Proc. International Symposium on Low Power Electronics and Design, Aug. 1998, pp. 293-298.

[11] S. Petit, J. Sahuquillo, J. M. Such and D. Kaeli, "Exploiting temporal locality in drowsy cache policies," Proc. ACM 2nd conference on Computing Frontiers, May 2005, pp. 371-377.

[12] M. Powell, Se-hyun Yang, B. Falsafi, K. Roy and T. N. Vijaykumar, "Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories," ISLPED 2000, pp. 90-95.

[13] M. Powell, Se-hyun Yang, B. Falsafi, K. Roy and T. N. Vijaykumar, "Reducing leakage in a high-performance deep-submicron instruction cache," IEEE Transactions on VLSI Systems, Vol. 9, Feb. 2001, pp. 77 – 89.

[14] S. Thoziyoor, N. Muralimanohar, J. H. Ahn and N. P. Jouppi, "CACTI 5 technical report, HPLabs," http://www.hpl.hp.com/ research/cacti/.

[15] R. Viswanath, V. Wakharkar, A. Watwe, V. Lebonheur, "Thermal Performance Challenges from Silicon to Systems," Intel Corp., Tech. Rep., 2000.

[16] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", Proceedings of the IEEE International Workshop on Workload Characterization, 2001, pp. 3-14.

[17] D.H. Albonesi, "Selective Cache Ways: On-Demand Cache Resource Allocation," Proc. IEEE MICRO, Nov. 1999, pp. 248-259.

[18] Chunho Lee, M. Potkonjak, W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", Proceedings of the IEEE/ACM International Symposium on Microarchitecture, (1997), pp. 330-335.

[19] B. Batson and T. N. Vijaykumar, "Reactive associative caches," Proc. International Conference on Parallel Architectures and Compilation Techniques, Sept. 2001, pp. 49-60.

[20] C. Brad, G. Dirk, and E. Joel, "Predictive Sequential Associative Cache," Proc. 2nd International Symposium on High-Performance Computer Architecture, Feb. 1996, pp. 244-253.

[21] A. Hasegawa, I.Kawasaki, K.Yamada, S.Yoshioka, S.Kawasaki, and P. Biswas, "SH3: High code density, low power," Proc. IEEE MICRO, Dec. 1995, pp. 11-19.

[22] K. Inoue, T. Ishihara, and K. Murakami, "Way-Predictive Set-Associative Cache for High Performance and Low Energy Consumption," Int. Symposium On Low Power Electronics and Design, 1999, pp. 273-275.

[23] J. Kin, M. Gupta and W. Mangione-Smith, "The Filter Cache: An Energy Efficient Memory Structure," Int. Symp. on Microarchitecture, Dec. 1997, pp. 184-193.

[24] M. Powell, A. Agarwal, T.N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing Set-Associative Cache Energy via Way-Prediction and Selective Direct-Mapping," Proc. IEEE MICRO 2001, pp. 54-65.

[25] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, J. Wolter, "The Performance of μ-Kernel-Based Systems" Proceedings of ACM Symposium on Operating System Principles, 1997, pp. 66-77.

[26] L. Soares, M. Stumm, "FlexSC: flexible system call scheduling with exception-less system calls" Proceedings of the 9th USENIX conference on Operating systems design and implementation, 2010, pp. 33-46.

[27] J. S. Hu, A. Nadgir, N. Vijaykrishnan, M.J. Irwin, and M. Kandemir, "Exploiting program Hotspots and code sequentiality for Instruction caches leakage management" Proceedings of ISLPED, 2003, pp. 593-601.

[28] Sung Woo Chung, K. Skadron, "On-Demand solution to minimize I-Cache leakage energy with maintaining performance" IEEE Transactions on Computers, Vol. 57, Issue 1, 2008, pp. 7-24.

[29] S.K. Srinivasan, M.N. Velev, "Formal Verification of an Intel XScale Processor Model with Scoreboarding, Specialized Execution Pipelines, and Imprecise Data-Memory Exceptions" Proceedings of the ACM and IEEE conference on Formal Methods and Models Co-Design, 2003, pp. 65-74.

[30] Intel, “The Intel XScale Microarchitecture” technical summary, available at: http://download.intel.com/design/intelxscale/XScaleDatasheet4.pdf, 2000.

[31] M. Alioto, P. Bennati, R. Giorgi, “Exploiting locality to improve leakage reduction in embedded drowsy I-caches at same area/speed” Proceedings of IEEE International Symposium on Circuits and Systems, 2010, pp. 37-40.

[32] J. Zushi, Gang Zeng, H. Tomiyama, H. Takada, K. Inoue, “Improved policies for drowsy caches in embedded processors” Proceedings of IEEE International symposium on Electronic Design, Test and Applications, 2008, pp. 362-367.

[33] C. F. Chen, S-H. Yang, B. Falsafi, A. Moshovos, “Accurate and Complexity-Effective Spatial Pattern Prediction” IEE Proceedings-Software, 2004, pp. 276-287.

[34] M.A.Z. Alves, Khubaib, E. Ebrahimi, V.T. Narasiman, C. Villavieja, P.O.A. Navaux, Y.N. Patt, “Energy Savings via Dead Sub-Block Prediction”, Proceedings of International Symposium of Computer Architecture and High Performance Computing, SBAC-PAD, 2012.