Low Cost Dynamic Architecture Adaptation Schemes for Drowsy Cache Management

Nitin Prakash¹,*, Israel Koren², and C. Mani Krishna²

¹Networking and Communication Division, IC Engineering, Cavium Inc., Marlborough, MA, 01752, USA
²Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA, 01375, USA

(Received: 12 June 2013; Accepted: 17 October 2013)

In this paper we develop low-cost dynamic architecture adaptation schemes to save leakage power in cache units. The proposed 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. Combining these schemes allows us to exploit more possibilities for energy reduction without incurring significant performance degradation.

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

1. INTRODUCTION

Energy consumption and speed of execution have long been recognized as conflicting requirements for processor design. The trade-off is especially important for embedded systems used in application areas 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. According to Ref. [5], the delay fault rate doubles for every 10 °C increase in the operating temperature.

In this paper, we consider the use of architecture adaptation to reduce power consumption without greatly affecting performance. Adaptation is complementary to other traditional schemes such as voltage and frequency scaling and can therefore be combined with them.

Leakage power is a major factor in the energy consumption of deep submicron circuits. Most of this leakage power originates in the caches, due to the presence of a large number of transistors. We therefore, target the on-chip cache units and attempt to reduce the performance penalty while still achieving lower energy consumption.

As applications become ever more complex with varying demands on caches, there is a need for dynamic architecture adaptation policies with minimal tuning requirements. In this work we have developed 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 static 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 a static selection of such intervals, we propose a low overhead dynamic adaptation scheme, where the interval is selected based on the runtime behavior of the application. Our dynamic scheme uses performance counters to manage drowsy caches. 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, and without imposing significant hardware overheads.

Earlier methods for drowsy caches (discussed in Section 2) assume that each row in the SRAM array has its own $V_{dd}$ line. However, modern SRAM arrays frequently share the $V_{dd}$ contacts amongst adjacent rows. Our design takes this into account and controls a pair of cache lines.

*Author to whom correspondence should be addressed.
Email: nitin.prakash@cavium.com
using a single voltage controller. This greatly reduces the associated design and area overhead, while providing a similar granularity of control as with dedicated $V_{dd}$ lines. To the best of our knowledge, this is the first attempt to implement a scheme of drowsy lines by taking into account shared $V_{dd}$ contacts.

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

We also propose a simple scheme for cache-way shutdown and integrate it with our drowsy cache scheme. 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.

The paper is organized as follows. In Section 2, we review the related work in the field, followed by our motivation in Section 3. Section 4 discusses the underlying circuit for implementing our approach, and the associated overhead. Section 5 presents our scheme for Icache and Dcache, and discusses the software overhead of the algorithm. Experimental results are presented in Section 6. In this section we determine the sensitivity of our dynamic schemes to its parameters, discuss some limitations of the static mechanisms and then compare our algorithm with previously known schemes. Section 7 discusses a combination of the drowsy line scheme with the Gated-$V_{dd}$ scheme. Section 8 proposes a combination of our drowsy line scheme with the cache-way shutdown mechanism. We demonstrate the effectiveness of our approach by using embedded applications from the MiBench and MediaBench suite. Section 9 provides a conclusion to our work.

2. REVIEW OF RELATED WORK

Over the past few years, several papers have considered architecture adaptation of caches. Zhang et al. have proposed a method called ‘way-concatenation’ to reduce dynamic power. With software control, they configure the cache to be 4-way associative, 2-way associative, or direct-mapped, to reduce dynamic power consumption. Furthermore, they have used the earlier known methods of shutting down cache ways to save leakage power. Nacul et al. have proposed a combination of Dynamic Voltage and Frequency Scaling (DVFS) and cache re-configuration where they dynamically vary the cache size, line size and associativity.

Designs such as gated-$V_{dd}$, ABB-MTCMOS (dynamically increasing threshold voltage), and voltage scaling have been proposed to control leakage power in transistors. Kaxiras et al. have used the gated-$V_{dd}$ method along with counters to control the drowsy cache lines. The counters measure the interval for which the respective cache line has not been accessed. 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-$V_{dd}$ scheme, the memory cell loses its data and hence an access to a drowsy line causes a miss. Similarly, Powell et al. using the gated-$V_{dd}$ 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. 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 between 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 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 alternative approach is presented, in which the predicted address is included in the tag array. However, this information is lost when the cache line is replaced, and hence not suitable for applications where miss rates are high.

Geiger et al. have proposed a combination of drowsy cache lines (with static intervals) and region-based caches. Petit et al. rely on reuse information to control drowsy caches. This mechanism is used for set-associative caches, wherein they keep awake only the Most Recently Used (MRU) line in every set. A comparison is made with the scheme where the two most recently used lines are kept awake in the set. 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. 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 6.7.2.

Alioto et al. 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 6.7.3.

Recently, researchers have proposed working at the granularity of sub-blocks within a cache line. This is based on the observation that not all the words in a cache line are accessed, especially if the cache line is large. Chen et al. 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. The next time the line is fetched, the sub-blocks that are predicted not to be accessed are switched-off (using the Gated $V_{dd}$ scheme). Alvez et al. have extended and modified the design to predict when the accessed sub-blocks become dead. A sub-block is deemed to be dead when the predicted number of accesses has been made to the sub-block. 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, 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-$V_{dd}$ scheme to implement it at a sub-block level.

Authors in Refs. [7, 18, 19] 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. In Ref. [18], the authors present the DHS (Dynamic Hot Spot 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 wakes up the next sequential line when an access is made. The work reported in Ref. [19] 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.

Re-configuring cache organization to save dynamic power was also investigated. Albonesi proposed to disable cache ways to reduce dynamic power consumption. Based on the allowed Performance Degradation Threshold (PDT), the applications are profiled to find the optimum number of cache ways that can be disabled. An AND gate is used to disable the access to particular cache ways. Albonesi 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.

Way predictive set-associative caches access one tag and data array initially (based on the prediction), and only access the other arrays if the initial array did not result in a match, yielding lower energy consumption at the expense of longer average access time. Prediction tables are used in Refs. [30, 32], and a tradeoff exists between the prediction accuracy and the quick availability of predicted address. In Ref. [31], the authors have used an MRU scheme for prediction. A comparison to this scheme is presented in Section 8.3.

3. MOTIVATIONAL EXAMPLES

We studied the general behavior of embedded applications with regard to cache accesses. We used 19 applications from the MiBench and MediaBench benchmark suites and recorded the cycles between successive hits to the cache. We observed that the cache access pattern is highly skewed. This is due to the fact that the applications have a large number of branches and control transfers. The access pattern is similar to the one shown in Fig. 1. Access pattern in 32 KB Icache.

![Access pattern in 32 KB Icache](image-url)
same cache line. Figures 1 and 2 show the results 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.

4. CIRCUIT IMPLEMENTATION

4.1. Additional Circuit for Leakage Reduction

To achieve reduction in leakage power, we have implemented drowsy cache lines using voltage scaling, as was done in Ref. [3]. This drowsy voltage is set to be approximately 1.5 times the threshold voltage \( V_T \). A typical value for the drowsy voltage is 0.3 V. We implement this drowsy mechanism only on 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 yields about 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 percentage increases with cache size.

The transition delay between the two voltage modes for a cache line depends on the size of the pass transistors in the voltage controller. According to Ref. [3], a 64\( \times \) \( \text{Leff} \) transistor has an access time of 1 cycle, while a 16\( \times \) \( \text{Leff} \) 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 shown in Ref. [3], the area overhead is 7.35\( \times \) equivalent_memory_cell per cache line (taking into consideration the bigger voltage controller and the circuit for the drowsy bit).

It must be noticed that Ref. [3] considers an individual voltage controller for every cache line. This means that the \( V_{\text{dd}} \) contacts cannot be shared by adjacent memory cells. The SRAM array area can be reduced by using a single voltage controller for a pair of adjacent cache lines. 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, of just 2 bits. 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 individual cache lines. For a particular cache line, if the local counter receives a signal from the global counter shortly after an access, the state is incremented immediately. Hence, the count in this scheme will not be accurate; the question is how well a scheme based on it performs in practice.

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 a 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 of the one-bit scheme 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
Prakash et al. Low Cost Dynamic Architecture Adaptation Schemes for Drowsy Cache Management

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

5. DYNAMIC SCHEME

5.1. 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 throughput. 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 experiments 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 at runtime and selects a near-optimal value of the drowsy interval. We first discuss our algorithm assuming that every cache line has a dedicated $V_{dd}$ line and voltage controller. In Section 5.4 we extend this scheme to the design where $V_{dd}$ 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 phase 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. The parameters of this algorithm, viz., repeat count and threshold ratio, are determined experimentally 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 Ref. [20], 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 the kernel mode is 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.

Reference [21] reports the size of the cache footprints for various system calls. The Icache pollution due to system calls is limited to a few tens of cache lines. 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 Dcache. No widely prevalent performance counters correlate 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 of 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 this purpose. 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, four instructions are fetched in every cycle (i.e., 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 an added delay on consecutive instructions. 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. We therefore, implement our dynamic scheme only for the Icache. In Section 6.4 the results when using dynamic and static schemes for the Dcache are compared.

5.4. Selecting Granularity-Sharing $V_{dd}$ Lines Across Adjacent Cells

As discussed above, adjacent rows in SRAM arrays generally share the $V_{dd}$ contacts. Controlling every row individually leads to a considerable area overhead. We can decrease the granularity of our control to reduce this overhead. We have a single voltage controller for each 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 cache lines pair if either of the two associated counters saturate.
2. Both Counters Saturate (BCS)—Switch off the cache lines pair 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 either 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.

6. EXPERIMENTAL RESULTS

6.1. Experimental Setup

We have used the Simplescalar tool\textsuperscript{2} to estimate the performance of our scheme. Sim-Wattch1.02d\textsuperscript{3} was used to simulate power. Cacti version 5.3\textsuperscript{4} was used to get the scaling factors for leakage in Dcache and Icache. These scaling factors have been included in the Sim-Wattch code to generate results for leakage power.

Our baseline processor is a superscalar with a fetch/issue/decode/commit width of 4. The size of the instruction fetch queue is 16. It includes 8-way set-associative, 32 KB Icache and Dcache. The combined L2 cache is 4-way and of size of 256 KB. The hit latency of the L1 (L2) cache is 1 (20) cycle(s). The main memory access time is 100 cycles for the first chunk and 6 cycles for each successive chunk of data. We use 45 nm technology for power modeling with an operating voltage of 1.2 V and frequency of 2 GHz. The threshold voltage is set at 0.2 V.

6.2. Performance Results of Our Schemes

We have simulated the performance of the two 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 3 shows the MIPJ benefits with respect to the base processor (with no drowsy caches). Figure 4 shows the IPC (Instructions per cycle) degradation with respect to the base processor.

As seen in the figures, the MIPJ benefits achieved by ECS are only slightly higher than those of BCS. The ECS scheme should provide more 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 its sensitivity 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 Energy × Time product with respect to the base processor. We individually check the effect of varying the IFR threshold ratio and the repeat count.

Figure 5 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 the repeat count is high, 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 and 5 works best. We choose the value of 5 for all our experiments.

Figure 6 shows the effect of varying the threshold ratio: we see that the optimal value of the threshold ratio is 0.8–0.9 for the benchmarks studied. 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.

We have also tried configurations where the learning cycle is forcibly restarted after a fixed number of cycles. This is because a phase change in the application might benefit from a different drowsy interval. Our method of relearning that is based on tracking IFR may not always be able to catch this. However, our simulations indicate that forcing relearning does not add any advantage to our technique, at least for the selected set of benchmarks.

For our final selection we use an IFR threshold ratio = 0.9 and repeat count = 5. We have verified these parameters with cache sizes of 16 KB, 8 KB and 4 KB showing a similar trend. The same set of parameters works satisfactorily for different cache sizes and associativity. An example is presented in Section 6.7.1.

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 either of these lines. The other line might well 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 parameters as those 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 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 7 shows the sensitivity to the direct overhead of system call, in terms of MIPJ. 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 8 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 a smaller 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 the direct overhead of system calls.

### 6.6. Our Dynamic Scheme versus Earlier Methods

As we have seen in Figures 3 and 4, the BCS scheme yields satisfactory results with lower performance

![Fig. 7. Sensitivity to direct overhead of system call.](image1)

![Fig. 8. Sensitivity to cache pollution overhead.](image2)

![Fig. 9. Comparison of MIPJ benefits (static and dynamic schemes).](image3)
6.6.1. Comparison with Static Scheme

In this section we compare our dynamic scheme to the static schemes presented in Ref. [3]. We use the NoAccess Policy from Ref. [3] which shows better results than the Simple Policy, for the static scheme. The Simple Policy puts all lines to drowsy mode when the drowsy interval elapses. The NoAccess Policy puts to drowsy mode only those lines that are not accessed within the drowsy interval. It uses a single bit per cache line to keep track of whether it was accessed in the last drowsy window. Unlike our scheme, it does not maintain a count of cycles elapsed since the last access to that particular cache line. For a fair comparison, we have modified the design in Ref. [3] so that it uses shared \(V_{dd}\) 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. We have calculated the 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 9 and 10 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 go down considerably.

We have also studied application behavior across cache sizes. Applications can show widely varying behavior for different cache sizes. Figures 11 and 12 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 achieved 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.

To demonstrate the robustness of the dynamic scheme, we show the results for Rijndael. We used our dynamic algorithm with the same parameter values as used for the 32 KB cache. Figure 13 shows that the dynamic algorithm is able to find the best configuration across cache sizes.

6.6.2. Comparison with Modified MRU (Most Recently Used) Scheme

We have implemented the dynamic scheme presented by Zushi et al.\(^{17}\) As mentioned in Section 2, this scheme is an improvement on the MRU scheme proposed by Petit et al.\(^{11}\) Along with the MRU information, the scheme...
in Ref. [17] 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 at 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 Ref. [17], like the others, assumes a dedicated \( V_{dd} \) for every cache line. For the sake of comparison, we have modified the design so that these \( V_{dd} \) 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 even if it does not qualify to be kept awake. As assumed in Ref. [26], we have used a window size of 4096 cycles. Figure 14 compares the MIPJ of our dynamic scheme (BCS) with that of the AOM and AAM schemes. Figure 15 compares the IPC degradation.

The dynamic BCS scheme clearly shows higher energy benefits for all applications with respect to the AOM scheme. The maximum difference in benefits is around 8% for \textit{Qsort}, and the average difference is more than 3%. The average performance degradation of the AOM scheme is lower 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 \textit{Rijndael} and 4.8% for \textit{Qsort}. The average performance degradation is almost the same for the BCS and the AAM scheme. Some applications like Basicmath, \textit{Fft} and \textit{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 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, 2-way associative cache, the MIPJ benefits of the AOM scheme reduces 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 a 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**

As mentioned in Section 2, Alioto et al.\(^2^2\) have proposed to 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 Improved Drowsy scheme,\(^2^2\) 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 16 compares the MIPJ of our BCS scheme to that of the Improved Drowsy scheme. This comparison has been made for the 8 KB direct mapped cache configuration used in Ref. [22]. Also, since we use the shared \(V_{dd}\) mechanism, we implement the Improved Drowsy scheme by controlling one pair of lines above and below any line.

As our simulations measure total energy consumption of the entire processor, we see that the MIPJ benefits are only around 1–2% for a 8 KB cache. 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 Ref. [22]. Some applications are used only in either of the two. The applications in Ref. [22] that have not been used in our simulations are Gsm, Pgp, Rynth, Typeset, Mad, Ispell. As presented in Ref. [22], 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. Also, with respect to IPC degradation, the dynamic BCS scheme performs better than the Improved Drowsy scheme for all applications. The average IPC degradation for the dynamic BCS scheme is 1.2%, while that for the Improved Drowsy scheme is more than 6%, with a maximum degradation of more than 16% for Rawcaudio.

We have observed that, for some applications like Fft and Stringsearch, the results presented in Ref. [22] do not match our simulations. The IPC degradation presented in Ref. [22] for these applications are much lower than shown here. This can be attributed to the different kinds of processors that have been used. We have used a 4-wide superscalar with an out-of-order pipeline implemented using Tomasulo’s algorithm. On the other hand, the authors in Ref. [22] have used an Intel Xscale processor.\(^2^6\) This processor issues one instruction at a time and uses three different pipeline stages after the RF (Register File) stage— the main execution stage, one for memory operations, and another for MAC instructions.\(^2^5\) Instructions are allowed to complete out-of-order. The dependencies are handled using Scoreboarding.\(^2^5\) Since, our processor has an issue width of 4, the penalty due to hitting a drowsy cache line is higher as the issue logic gets adversely affected if the Instruction Fetch Queue has fewer instructions. Furthermore, the XScale processor uses Scoreboarding, which stalls for both RAW and WAW hazards. 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 45 nm technology. In this section, we compare these to 32 and 22 nm technology nodes. Figure 17 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 due to the scaling down of the \(V_{dd}\). 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 100 nA/\(\mu\)m. Due to the reduction in \(V_{dd}\), the scope for voltage scaling reduces by some amount. However, we are still able to get meaningful benefits even for 22 nm...
technology. Since no changes have been considered to the design, the IPC results remain the same.

7. COMBINING DROWSY CACHE LINES AND GATED-\(V_{dd}\) SCHEMES

The drowsy cache scheme scales down the voltage only to a point where the data in the memory cell is not lost. On the other hand, the Gated-\(V_{dd}\) mechanism completely turns off the cache lines. This results in the data being lost but the leakage energy savings are higher than 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,\(^1\) and that for the Gated-\(V_{dd}\) scheme.\(^1\) The Gated-\(V_{dd}\) scheme introduces a high-\(V_{t}\) 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 all power to the SRAM cell. As this is a high-\(V_{t}\) transistor, its leakage is minimal but its access time is high. 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-\(V_{dd}\), dual-\(V_{t}\) transistor, which can provide 97% leakage benefits without any increase in access time, and the resulting area overhead is 5%.\(^1\)\(^\text{12}\)

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 switch them off completely if these drowsy 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 saturating 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 since we have used a static drowsy interval of only 100 for the BCS scheme and switching off the lines after another 100 cycles considerably increases the performance degradation. 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 but use a different technique to implement it. Since the global counter for the Dcache provides increment signals corresponding to a drowsy interval of 100, a very large local counter would be required to count 8000 resulting in a prohibitive overhead. 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. When the drowsy bit is 1, the local counter uses increment signals from the other global counter.

7.3. Performance Results

We have implemented the proposed scheme with the parameters discussed above. Figure 18 compares the MIPJ benefits, when using only the drowsy cache scheme...
(dynamic BCS), with that when using a combination of drowsy cache and Gated-\$V_{dd}\$ schemes. The results show a mix of applications that benefit from the scheme and those who do not. The benefits are not very high indicating that the added advantage of Gated-\$V_{dd}\$ scheme with respect to leakage benefits is not very high. The drowsy voltage scheme provides around 78% leakage reduction while the Gated-\$V_{dd}\$ scheme provides 97% (for individual SRAM cells). However, as discussed before, these schemes have only been implemented on the data array of the cache. Furthermore, the Gated-\$V_{dd}\$ 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.

The authors of Ref. [33] have developed a scheme for timing aware leakage control for hard real-time systems. They have analyzed a combination of Gated-\$V_{dd}\$ scheme and static Drowsy Interval scheme for their algorithm. They too conclude that the Gated-\$V_{dd}\$ scheme does not provide enough benefits to warrant the design overhead.

8. COMBINING DROWSY CACHE LINES AND CACHE WAY SHUTDOWN SCHEMES

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

Albonesi has proposed a way-selection scheme for saving dynamic power.\(^{29}\) We further switch off the cache ways to which access is disabled, using the gated-\$V_{dd}\$ mechanism,\(^{12}\) thereby saving leakage power also. This mechanism should not be confused with the one presented in the previous section. In this scheme, the voltage to the entire cache way 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,\(^{29}\) 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. Still, 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. Based on our sensitivity analysis, a learning interval between 3000-5000 cycles is optimal. A threshold ratio of 0.9 is used for the miss rates.

8.2. Integrating Cache-Way Shutdown and Drowsy Cache Line Schemes

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. Learning for deciding on the cache ways is based on the tracking of miss rates, while learning for drowsy interval for Icache is based on the IFR. 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 IFR.

8.3. Performance Results and Comparison to Earlier Schemes

We have seen a considerable increase in the MIPJ benefits. As compared to using only drowsy lines, the average benefit has increased from 18.27% to 32.12%, while the average performance degradation has increased from 1.2% to 3.03%. It should be noted that MIPJ benefits are defined as ((MIPJ of processor with our scheme-MIPJ of Base Processor)/MIPJ of Base Processor). It represents the same idea as Energy benefits, but the numbers for Energy benefits and MIPJ benefits can differ widely as the Base processor has higher Energy consumption but lower MIPJ value.

8.3.1. Comparison to the Predictive Cache Way Scheme

Our first comparison is with the predictive cache way scheme discussed in Ref. [31]. MRU information is used to predict the cache way in this design. In the first phase, only this cache way is accessed. Only in the event of a miss are the other cache ways accessed. Hence, in the case of a hit on the predicted way, dynamic energy is consumed for that cache-way only. In case of a misprediction, the energy consumption is the same as that of a standard cache. As discussed in Ref. [31], we assume that the getting the MRU information does not increase the cache access time. This can be achieved by calculating the
set-index address at an earlier stage in the pipeline. As in our scheme, we have assumed a cache hit time of 1 cycle. This is the hit time if there is a hit on the predicted way. In case of a misprediction, the other cache ways need to be accessed and the cache hit time increases to 2 cycles. It should be noted that, this design provides only dynamic energy benefits.

Figures 19 and 20 compare the MIPJ and IPC, respectively. Our dynamic scheme for the combination of cache-way shutdown and drowsy lines clearly outperforms the predictive cache in terms of MIPJ. With respect to performance degradation, our dynamic scheme performs worse only for a few applications like Tiffmedian, Mpeg2dec, Qsort and H263enc. The average performance degradation of our scheme is still lower (3.03%) when compared to that of the predictive cache (5.44%).

8.3.2. Comparison with the DRI-Icache Scheme

Our second comparison is to the DRI-Icache (Dynamically Resizable Icache) presented in Refs. [12 and 13]. Instead of cache ways, reconfiguration is done by changing the number of sets in the cache. Based on a threshold placed on the miss rate, the sets are switched off using the Gated-$V_{dd}$ mechanism. This helps in saving dynamic as well as leakage power. However, changing the number of sets requires modifications in the way the address is decoded. In case of downsizing, fewer index bits are used and more bits are needed for tag comparison. By using the right number of bits, the correct data can be accessed in the sets that are alive. However, after an upsizing, a previously existing block may be mapped to a different set (since more index bits are used). In an Icache, this kind of aliasing will increase the miss rate but for the Dcache, incorrect data might be accessed in case of dirty blocks. Due to this complication of handling dirty blocks, the authors have only implemented this scheme in an Icache.

For the sake of comparison, we too have implemented our design only on the Icache. Both designs use a conventional Dcache. Reference [13] has four different parameters that need to be set for their algorithm. The miss-bound is the threshold on the miss rate. The size-bound is the minimum size the cache can attain. Sense interval is the fixed period after which the miss counts are measured and configurations are updated. Divisibility is the speed at which the cache size is reduced. We have used the parameter values as presented in Refs. [12 and 13]. The sense interval is set at 1 million instructions. Divisibility is set at 2, that is, the cache is downsized or upsized by a factor of 2. The authors propose to use different values of miss-bound and size-bound for all applications, which need to be found out by profiling. In contrast, our algorithm strives to find common values for its parameters across various kinds of applications. For this comparison we have used the size-bound as 4 K (same as the minimum size in our design). We present the comparison results with different values of miss-bound. Furthermore, Ref. [13] proposes a scheme to prevent constant switching between two configurations. In case the optimal size of the cache for an application is in between two configurable sizes, continuous resizing takes place between two adjacent sizes, after every sense interval. In such a situation, a 3-bit saturating counter is used to detect it, and downsizing is prevented for 10 consecutive sense intervals.

Figures 21 and 22 compare the MIPJ and IPC, respectively. Except for Cjpeg and H263enc, our algorithm works equal to, or better than the DRI-Icache. Decreasing the miss-bound below 1 K reduces the scope for reduction in the DRI-Icache size for most applications and gives lower
energy benefits. Increasing the miss-bound continuously, leads to the selection of very small cache sizes, which causes drastic degradation in a few applications. With the DRI-Icache, few applications show degradation in MIPJ with all three selections of the miss-bound. This is mainly due to the fact that the configurations are switching between two cache sizes. The throttling mechanism only stops this for 10 sense intervals at a time, after which the toggling starts again. The behavior becomes exponentially worse for larger miss-bounds where smaller cache sizes are selected and the cache starts thrashing. Selecting a proper size bound, for example, 32 KB for Basicmath and Rijndael, can rectify this behavior, in which case there will be no energy benefits. Hence, the DRI-Icache scheme is largely dependent on individual selection of parameters for every application.

9. CONCLUSION
Large caches have the potential to waste considerable energy due to leakage. A drowsy scheme is effective in reducing such losses; however, 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 satisfactorily with caches of different associativity and sizes. With our dynamic scheme, there is no longer a need to profile individual applications across various datasets, as would have been necessary with the static schemes. Compared to earlier published schemes, our dynamic scheme provides more energy benefits with lower performance degradation. The achieved MIPJ benefits are 18.27% with an IPC degradation of 1.2%. We have also proposed a combination of our drowsy line scheme with the cache-way shutdown mechanism, to further increase energy benefits.

Acknowledgment: This work was supported in part by the National Science Foundation under grant CNS-0931035.

References


Nitin Prakash
Nitin Prakash received his B.S. degree in Electronics and Communication Engineering from Manipal Institute of Technology, India, in 2008 and his M.S. degree in the Computer Engineering from University of Massachusetts, Amherst, USA, in 2013. He has previously worked as a compiler engineer with Toshiba, India and is currently working as a hardware engineer at Cavium Inc., Marlborough, USA. His interests include computer architecture, computer arithmetic, and embedded hardware and software.

Israel Koren
Israel Koren is currently a Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst and a Fellow of the IEEE. He has been a consultant to numerous companies including IBM, Analog Devices, Intel, AMD and National Semiconductors. His research interests include Fault-Tolerant systems, Computer Architecture, VLSI yield and reliability, Secure Cryptographic systems, and Computer Arithmetic. He publishes extensively and has over 250 publications in refereed journals and conferences. He is an Associate Editor of the VLSI Design Journal, and Sustainable Computing: Informatics and Systems. He served as General Chair, Program Chair and Program Committee member for numerous conferences. He is the author of the textbook “Computer Arithmetic Algorithms,” 2nd Edition, A. K. Peters, 2002 and a co-author of “Fault Tolerant Systems,” Morgan-Kaufman, 2007.

C. Mani Krishna
C. Mani Krishna is a professor of Electrical and Computer Engineering at the University of Massachusetts at Amherst. He received his Ph.D. from the University of Michigan. His areas of interest include real-time systems, fault-tolerant computing, cyber-physical systems, and system performance evaluation. He has coauthored textbooks in real-time systems and fault-tolerant computing.