ABSTRACT

Title of Thesis: RUN-TIME INSTRUCTION CACHE CONFIGURABILITY FOR ENERGY EFFICIENCY IN EMBEDDED MULTITASKING WORKLOADS

Degree Candidate and Year: Mathew Paul
Master of Science, 2008

Directed By: Professor Peter Petrov,
Department of Electrical and Computer Engineering

In this thesis we propose a methodology for energy reduction in multitasking computing systems by addressing the energy consumption of the on-chip instruction cache. Our technique leverages recently introduced reconfigurable cache technology to partition the instruction cache at run-time using application specific profile information. Each application is given a sub-section of the cache as its partition which alone is kept active while the corresponding application is executed. The remaining inactive sections are kept in a low-power mode, reducing both dynamic and leakage power. Isolating tasks into disjoint cache partitions also results in eliminating or drastically reducing inter-task I-cache interference. No prior information about the

timing of the tasks within the workload is required. In some cases, partitions may be required to overlap, which could degrade performance because of cache interference in the overlapped region. For such cases we propose and evaluate run-time partition update policies which trade-off the power savings to ensure guaranteed performance.

RUN-TIME INSTRUCTION CACHE CONFIGURABILITY FOR ENERGY EFFICIENCY IN EMBEDDED MULTITASKING WORKLOADS

By

Mathew Paul

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Master of Science
2008

Advisory Committee:
Professor Peter Petrov, Chair
Professor Shuvra Bhattacharyya
Professor Manoj Franklin

© Copyright by
Mathew Paul
2008

Dedication

To God.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Dr. Peter Petrov, for his guidance and patience throughout the course of this research. His approach towards tackling hard problems motivated me in several difficult situations and kept me going till the end.

I would also like to thank Dr. Shuvra Bhattacharya and Dr. Manoj Franklin for readily agreeing to serve on my thesis committee. I greatly value their support and encouragement.

I am also thankful to Rakesh Reddy for patiently answering my many queries related to his previous work on inter-task cache interference in data caches. A special thanks to my friends at home for putting up with my occasional eccentricities during this period.

Finally, my family has always been beside me in all my endeavors. I would like to thank my parents, Paul and Mercy, for their support and prayers. A special note of thanks to my brother, John, who spent time with me in discussing several issues related to the research, especially the coding of the simulator. I shall always be grateful to them for their sacrifices and urging me on towards the finish line.

Table of Contents

Dedication……………………………………..……………………………..ii
Acknowledgements…………………………..…..………………………...iii
Table of Contents……………………………………..……………………..iv
List of Tables……..……………………………………..………...………..v
List of Figures…..…………………………..……..……..………………...vi
Chapter 1: Introduction………………………..……………………………..1
1.1 Thesis Contributions.. 5
1.2 Challenges………………………………………...6
1.3 Cache partition representation.. 7
1.4 Outline of this thesis...8
Chapter 2: Related Work…...…...……………………………………….……9
2.1 Low Power Cache 9
2.2 Reconfigurable caches..10
2.3 Inter task cache interference 12
Chapter 3: Cache Configurability………………………………………...…14
3.1 Configurable cache architectures 14
Chapter 4: Cache Partitioning………………………………………………...19
4.1 The Benchmarks 19
4.2 Partition Methodology...22
4.3 Partition Update 27
4.4 Partition reshuffle...29
Chapter 5: Evaluation environment…………………………………………...32
5.1 The Simulator 32
Chapter 6: Experimental Results…………...…………...…………………………..36
6.1 Impact on performance..36
6.2 Impact on power 40
Chapter 7: Conclusion……………………..…………………………………..48
Bibliography………………………………..…………...……………………..49

List of Tables

Table 4.1: Dynamic Benchmarks……………………………………………………….…21
Table 4.2 Static benchmarks ………………………………………………………...…21

List of Figures

Fig 1.1 Miss rate and normalized energy of EPIC and MPEG2 on 8kB data caches of different associativities……………………………………………………………..4

Fig 1.2 Miss rate of EPIC for different Instruction cache configurations…………….5

Fig 1.3 Cache partition representation………………………………………………..7

Fig 3.1: Way shutdown cache………………………………………………………..15

Fig 3.2: Way concatenation cache…………………………………………………...17

Fig 3.3: Drowsy cache line circuitry ………………………………………………...19

Fig 4.1: Timeline of a static benchmark…………………………………………..…20

Fig 4.2: Timeline of a dynamic benchmark…………………………………………20

Fig 4.3: Miss rates plots of individual benchmark tasks……………………………22

Fig 4.4: Exploration of partition space………………………………………………22

Fig 4.5: Cache partitioning for a 3 task benchmark..………………………………...26

Fig 4.6: Heuristic partition update algorithm………………………………………...29

Fig 4.7: Cache partition reshuffling…………………………………………………31

Fig 5.1: 32kB, 4way cache modeled using 64 independent 512 byte caches………..32

Fig 5.2: Allocation and overlap of cache partition…………………………………..33

Fig 5.3: Way usage record for modeling multiple cache ways……………………....34

Fig 5.4: Block diagram representation of complete experimental setup…………....35

Fig 6.1: Miss rate comparison for dynamic benchmarks for 16kB baseline cache …37

Fig 6.2: Miss rate comparison for static benchmarks for 16kB baseline cache…….38

Fig 6.3: Miss rate comparison for static benchmarks for 32kB baseline cache……..38

Fig 6.4: Miss rate comparison for dynamic benchmarks for 32kB baseline cache….39

Fig 6.5: Percentage reduction in dynamic energy for dynamic benchmarks with 16kB baseline cache for 70 nm technology……………………………………………..41

Fig 6.6: Percentage reduction in dynamic energy for static benchmarks with 16kB baseline cache for 70 nm technology..………………………………..…………..41

Fig 6.7: Percentage reduction in dynamic energy for dynamic benchmarks with 32kB baseline cache for 70 nm technology..……………………………………………42

Fig 6.8: Percentage reduction in dynamic energy for static benchmarks with 16kB baseline cache for 70 nm technology...……………………………………………42

Fig 6.9: Average cache utilization………..………………………………………43

Fig 6.10: Percentage reduction in static leakage power for dynamic benchmarks with 16kB baseline cache for 70 nm technology………………………………………44

Fig 6.11: Percentage reduction in static leakage power for static benchmarks with 16kB baseline cache for 70 nm technology....……………………………………44

Fig 6:12: Percentage reduction in static leakage power for dynamic benchmarks with 32kB baseline cache for 70 nm technology....……………………………………45

Fig 6.13: Percentage reduction in static leakage power for dynamic benchmarks with 16kB baseline cache for 70 nm technology...……………………………………..45

Fig 6.14 Percentage reduction in static leakage power for dynamic benchmarks with 16kB baseline cache for 0.18um technology……………………………………...46

Fig 6.15 Percentage reduction in static leakage power for static benchmarks with 16kB baseline cache for 0.18um technology....……………………………………...47

Fig 6:16 Percentage reduction in static leakage power for dynamic benchmarks with 32kB baseline cache for 0.18um technology....……………………………………...47

Fig 6.17 Percentage reduction in static leakage power for dynamic benchmarks with 16kB baseline cache for 0.18um technology...47

Chapter 1: Introduction

Embedded systems are special-purpose computer systems designed to perform one or a few dedicated functions, often with very limited power budget and real-time computing constraints [25]. In the early days these were widely used in simple applications like traffic lights, watches and other control devices. These days embedded systems comprise smart phones and other hand held devices such as media players, digital cameras etc. The growing importance of multimedia in the present day world has increased manifold the performance requirement on embedded systems. For example, most of the modern mobile phones have built in camera, video recorder, speech recognition software, streaming television etc, in addition to the other wireless capabilities. The computing system which is at the heart of these devices has to manage all these applications while satisfying performance, power, and real-time requirements as well. Similar to general purpose computers, embedded systems have to handle multiple concurrent tasks, but with limited resources. This has lead to many features being borrowed from the high performance general purpose computing domain. The use of caches and the need for multitasking being the major features borrowed by embedded systems.

A cache is a comparatively smaller on-chip memory placed in between the main memory and the processor. It employs the principal of locality, both in time and in space, to increase the memory bandwidth and thus reduce the ever widening bandwidth gap between the processor and the memory. Processor speeds have increased several times over in the last few years, but memory systems have not been

able to maintain a similar increase in their speed. Mostly due to the structure and manufacturing techniques, the main memory is several times slower compared to the processor. Hence for increased utility of the processor there is a need to increase the memory bandwidth by introducing a memory hierarchy with faster smaller memories closer to the processor. This is the fundamental goal of caches. The underlying principle is that if a certain memory location is accessed there is a high probability that it will be accessed again in the near future, i.e. locality in time. So by retaining it in the cache memory for some time, future accesses to this memory location can be serviced from the faster cache instead of from the slower memory downstream. Also if a certain memory location is accessed there is also high probability that other memory locations near it will also be accessed in the near future. This is locality in space. Hence data is loaded into the cache memory in chunks, usually referred to as a cache line or a cache block. Initially real-time embedded systems avoided having a cache in place to provide for better predictability of the applications with real-time constraints. The presence of a cache complicates the process of estimating the worst case performance of these applications because of the possibility that instructions could miss in the cache leading to a pessimistic bound. But with the increased memory bandwidth requirements due to the many multimedia applications such as video encoders/decoders, speech codecs etc, modern embedded systems are forced to have on-chip caches. These are implemented as one or two levels of on-chip caches usually referred to as L1 and L2 caches.

The on-chip cache occupies a major portion of the IC area and consists of densely packed transistors. Ranganathan et al. in [13] indicate that cache memory can occupy

up to 50% of the die area and consume up to 80% of the transistor budget. On-chip cache is hence a major contributor to the total energy consumed by the chip. For hand held devices battery life is a major design issue and longer battery life translates to higher quality of the product. Some of the research on reducing power consumption has hence focused on low-power caches to reduce the energy consumed in these embedded systems. The execution patterns and how embedded applications utilize the cache memory has been extensively studied and results have shown scope for optimizing the cache for the applications at hand. In [13] Ranganathan et al. underline the streaming data access patterns and large working sets of multimedia applications and how they are ineffective in using the large caches. The ineffectiveness of multimedia applications in making full use of the on-chip cache is also highlighted in the work by Zhang et al [19] where they state that increasing the associativity ways of the cache may not directly map to improved performance for many multimedia applications. Most tasks perform reasonably well with one or two cache ways. A direct mapped cache consumes less power than a four way set associative cache since only one tag and one data array are read instead of four tag and data arrays. Fig 1.1 shows, for example, how the performance of EPIC and MPEG2 are only minimally influenced by increasing the number of cache ways of the data cache beyond two. With four cache ways the energy consumption increases for no significant increase in performance. Based on these observations, energy reduction techniques in research on low power cache architectures have focused on tuning the cache to a particular application. This increased interest in tuning the cache architecture has spawned a body of research on configurable caches [1][2][3][15][19].

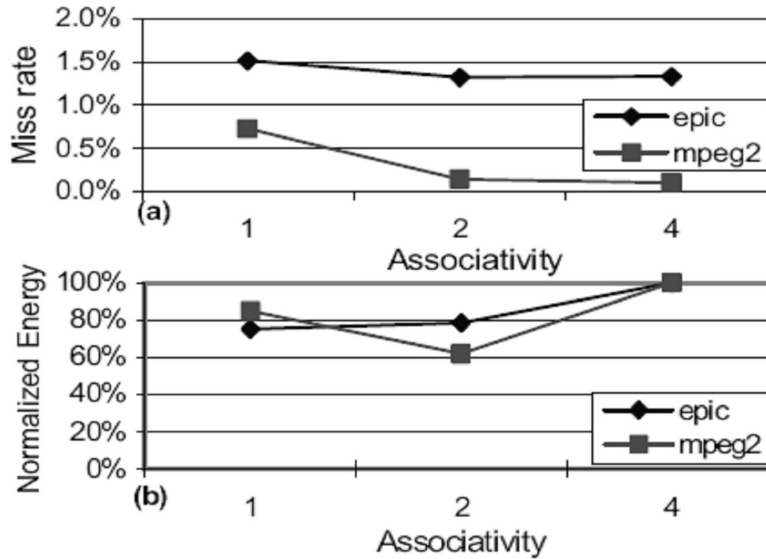


Fig 1.1: Miss rate (a) and normalized energy of EPIC and MPEG2 on 8kB data caches of different associativites. [19]

With the advent of embedded multitasking, the different tasks are required to share the cache. In [22] Agarwal et al. demonstrate that multiple tasks sharing the cache by using unique process identifier (PID) for each process is more effective than flushing the cache on context switch to prevent inadvertent use of cache data. Multiple tasks sharing the cache, but causes the problem of inter-task cache interference which degrades the performance of individual tasks by causing cold misses around the point of context switch. The currently active task can evict data previously loaded into the cache by a task which was preempted and swapped out of context. When this task is scheduled by the operating system again, it may miss in the cache resulting in cold misses which would not have occurred had there been no other interfering task. These misses are classified as inter-task cache interference misses.

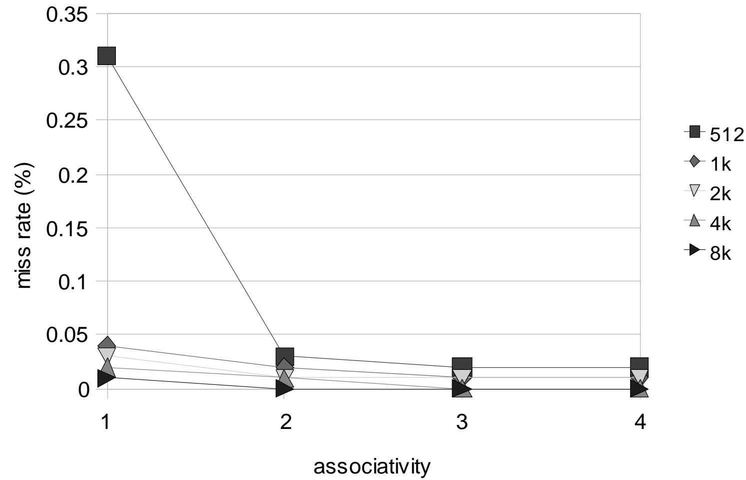


Fig 1.2: Miss rate of EPIC for different instruction cache configurations.

If a large cache, say a 16kB 4way set associative instruction cache, is used in an embedded system, the ineffectiveness of multimedia applications like EPIC in fully utilizing the cache has already been mentioned. With the problem of inter-task cache interference, EPIC, which by itself would require only a small cache for good performance (see Fig 1.2), would start populating the entire cache and in the process interfere with other applications which may require much larger cache and degrading their performance.

*1.1 **Thesis Contributions***

Based on these observations we propose a partitioning scheme which leverages reconfigurable cache architecture [1][3][15] and partitions the instruction cache to minimize the energy consumed by it and at the same time keep the performance of each task within predefined thresholds. Previous work in this area by Reddy et al. [2] focused on statically partitioning the data cache between a known set of tasks and defining a partitioning methodology which partitions the cache offline. Our work

focuses instead on partitioning the instruction cache at run-time and requires no prior information of the timing of each task. It only makes use of static profile information of each task to determine its optimum cache partition. For example, if there are three tasks running concurrently in a time-shared manner, we determine the required cache space for each task, based on some performance thresholds, and map them to a portion of the cache which we call the cache partition for that task. By keeping only the currently executing task's partition active, dynamic and static power are significantly reduced. Also, providing each task its own partition with no overlap with others, helps avoid cache interference and hence the performance of each task is close to the performance obtained when all tasks share the baseline cache. In cases of partitions overlapping additional techniques are proposed and evaluated to update partitions so to achieve guaranteed performance.

1.2 Challenges

Cache partition overlap cannot be avoided in all cases. Some tasks require larger cache and when there are many tasks running concurrently, not all partitions can be allocated exclusive of each other. In such cases partitions are allowed to overlap. Applications with overlapped partitions may exhibit performance which may exceed the thresholds due to possible inter-task cache interference. Hence techniques are required to suitably modify cache partitions at run-time to maintain performance within thresholds.

System level support is required for providing cache miss statistics in the task binary. At compile time the task binary is populated with the task's cache miss

statistics and stored in a tabular form. The Operating system needs to be supported with additional capability to look up the task binary and determine the appropriate cache partition.

Traditional trace driven cache simulators like dineroIV [26] can only simulate single cache instance. Simulator with support for multiple cache partitions and allowing overlapped partitions has to be developed.

1.3 Cache partition representation

In subsequent sections we use the following representations for baseline cache, cache ways and individual partitions.

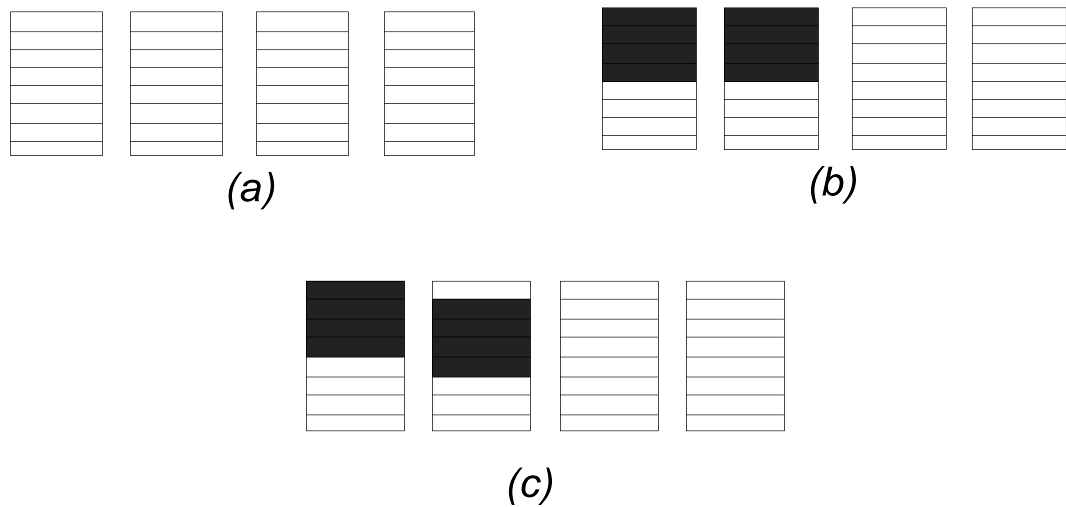


Fig 1.3: cache partition representation.

The baseline instruction cache is represented by four rectangles placed next to each other, as shown in Fig 1.3 (a), with each complete rectangle representing one associativity way of the baseline cache. Unless mentioned each cache way is 4kB; hence the baseline cache of Fig 1.3 (a) is a 16kB, 4 way set associative cache. The smaller rectangular blocks within each cache way represent 16 contiguous cache

lines. For a cache line size of 32 bytes, the smaller rectangular blocks correspond to 512 bytes of cache memory. A cache partition allocated to a particular task is a collection of these smaller blocks, subject to some validity criteria. Partitions can be within one cache way or across multiple ways depending on the associativity ways for that partition. For a cache partition to be valid the number of sets within each of its cache ways must be a power of 2. When cache partitions extend over multiple ways, corresponding cache lines from each cache must be selected to form the partition as shown in fig 1.3 (b) for a 4kB, 2way cache. Misaligned collection of cache lines like the one shown in Fig 1.3 (c) do not form a valid partition.

1.4 Outline of this thesis

The remainder of the thesis report is structured as follows. Chapter 2 looks at some of the recent works in related areas like low-power caches, reconfigurable cache architectures, and the phenomenon of cache interference. Chapter 3 explains in some detail the architectural modifications suggested for reconfigurable caches. The partitioning algorithm is explained in detail in chapter 4. The challenges associated with cache partition overlap and how they are effectively handled by our *partition update* and *partition reshuffle* policies are also included in chapter 4. Chapter 5 explores the evaluation environment detailing the development of our simulator. The experimental results and the evaluation of the algorithm in terms of the energy savings achieved are analyzed in chapter 6. Finally, Chapter 7 concludes the thesis report.

Chapter 2: Related Work

2.1 Low Power Cache

With the increasing on-chip cache sizes and switching frequency, the energy consumed by these caches has increased drastically. Researchers have been proposing several techniques to cut down the energy consumed by the on-chip caches through various prediction methods or accessing only sections of a set associative cache. In [8] the author proposes using the most recently used (MRU) algorithm to predict the cache way where a hit is possible for each set in the cache. For a 4 way set associative cache, two additional bits for each set would suffice to keep track of the MRU cache way. Following this method for each access, rather than look up the data and tag array for each of the four cache ways, only the predicted way is checked for a hit. In case of a misprediction, the remaining three ways can be looked up. If the prediction hit rate is good this scheme reduces energy consumption by close to 80% and also reduces access time by 50%. The scheme was able to achieve average prediction hit rates of 96% and 86% for I and D caches respectively. Auxiliary cache based memory hierarchy schemes [9] [20] [21] have also been proposed which introduce a smaller faster cache between the processor and the larger, power inefficient L1 cache. The large number of transistors devoted to the cache memory is the major reason for the high power consumption of the on-chip caches. Hence [10] propose the use of a much smaller Filter Cache as an additional level between the processor and the L1 cache. The requests from the CPU are handled by the Filter Cache and the L1 cache can be put in a low power mode when the data is available in the Filter Cache. This provides

for considerable power savings as the L1 cache needs to be powered on only in case of a miss in the Filter Cache. The energy savings come at the expense of performance degradation due to the slightly increased cache access time with the addition of the Filter Cache into the critical path. To correct this performance degradation, in Predictive Filter caches [11] the instructions are accessed from the L1 cache or from the filter cache based on whether the predictor predicts a hit in the filter cache. Based on the effectiveness of the prediction, the extra cycles in the case of a miss in Filter Cache can be avoided by directly accessing the main cache.

There has also been specific focus on the instruction cache and techniques to reduce power consumption have focused on the tag comparisons in the I-cache [12]. Instruction cache is mostly accessed in a sequential manner except when a control transfer (branch, call) instruction is executed, in which case execution continues from the target instruction. Since the tag remains the same for all addresses in the same cache line, sequential instructions within the same cache line need not all have the tag compared for each instruction. This elimination of tag comparison helps reduce the power consumed in the I-cache.

2.2 Reconfigurable caches

As seen from previous discussions, there is need to reduce the energy consumed by on-chip cache and some techniques have been proposed in the previous section. Another approach to reduce the energy consumed by the cache memory is to use configurable caches and customizing the cache for the particular applications at hand. The work in [13] underlines the importance of customizing the cache for various applications because different applications exhibit different cache requirements.

Having a single large cache might lead to majority of the cache being unused by several applications and as a result increased energy usage without much contribution to performance improvement. In their research they propose a reconfigurable cache design which allows the cache SRAM arrays to be dynamically divided into multiple partitions that can be used for different processor activities. The idea of using only sections of the cache for some applications was further developed by the work of Albonesi [1] where an additional cache way select register (CWSR) is used to indicate which ways would remain active and which do not. Shutting down the cache ways decreases the dynamic power consumed but can increase the cache misses because of the reduction in the overall cache size. Performance Degradation Threshold (PDT) is set for each application so as to shut down ways under the guarantee of certain performance levels. The fact that set associative caches are implemented as subarrays allows selective cache ways to be implemented with minimum hardware overhead. In [19] the authors propose the use of a configuration register to allow the flexibility to configure the number of ways of the baseline cache as required by the application. The technique referred to as way concatenation builds on the way shut down cache proposed by Albonesi [1]. By setting the two one bit registers, reg0 and reg1 to 0 or 1, the number of associativity ways of the cache can be varied as 4, 2 or 1. The least two significant bits of the tag part are decoded along with reg0 and reg1 to activate one or more of the cache ways. Energy savings between 40%-60% are reported for the configurable cache depending on the number of ways kept active. The energy saving come from the fact that fewer number of sense amplifiers, bit lines and word lines are kept active per access when these caches

are configured for lesser number of ways. The research also made use of the gated-V_{dd} circuit level technique proposed by Powell et al. [14] to further decrease the leakage current. The way concatenation effectively reduced the dynamic power by activating only part of the entire cache. The addition of the extra transistor in the gated-V_{dd} technique helped reduce the leakage current through the stacking effect of the self reverse-biasing series-connected transistors. The cache reconfiguration can also be done by reducing the effective number of sets within each cache way [15]. In the Dynamically Resizable Instruction Cache (DRI I-cache), decreasing the cache size means lesser number of index bits are required to determine the particular set within the cache way. For this, a section of the index portion needs to be masked. To allow for dynamically varying the number of sets, a mask is maintained in a register and shifted right in case of decreasing cache size and let to increase the number of sets. For the benchmarks evaluated, the DRI I-cache was successful in reducing the cache size on an average by 62% while increasing the execution time by less than 4%. Zhang et al. in [23] describe by using counters to track the miss rate of each application, how the reconfigurable cache architectures mentioned above can be used to fine tune the cache for each application.

2.3 Inter task cache interference

The cache-performance costs of a context switch may be greater that all other context-switch costs. In [4] Mogul et al. indicate that a context switch wastes several thousand instruction cycles on average due to the phenomenon of inter-task cache interference, which is comparable to the time it takes to send or receive a network packet. Agarwal et al. [22] studied the effect of multitask workload on cache

performance by using a trace driven cache simulator. Cache interference was studied by comparing miss rates of applications running alone versus together for different cache configurations. The work shows that flushing the cache on each context switch to avoid inadvertent use of data performs worse that use PID in a virtual cache and letting the multiple tasks share the cache. Lot of research has gone into proposing different techniques for reducing or in some cases eliminating the detrimental effects of inter task cache interference. One class of techniques partitions the available cache between the various applications and only allow each application access to it's own partition. This can be achieved through additional hardware support or through software. In [5] the authors propose extra address-mapping hardware to map the access of each application to it's own cache partition. The cache is partitioned into equal-sized portions for each task and a larger partition, the shared pool, which is shared by the non real-time tasks and also used for synchronization between real-time tasks. Software based cache partitioning schemes on the other hand use special compiler and linker support to map the address to the partition assigned to the particular application [6]. For preemptive cached real-time systems, [7] proposes a software based partitioning scheme which both eliminates the inter task cache interference and also improve the predictability of real-time applications. The special compiler breaks up any linear code or data segment into non-linear memory partitions each of which map into the cache partition assigned for that particular task.

Chapter 3: Cache Configurability

3.1 Configurable cache architectures

Different applications exhibit different cache usage patterns and hence tuning the cache configuration to that of the current active application improves the usage factor. If the application does not use some of the associativity ways, the unused ways can be shut down or placed in a low power mode to save on power consumption. Reconfigurable caches provide additional hardware support to enable and disable sections of the cache on a need basis with the aim of reduced power consumption.

The power consumed by the on chip memories can be classified into two major categories: Dynamic power and Static Power. Dynamic power is the power consumed at the time of switching the on chip transistors from one state to another and is used up in charging and discharging the capacitive loads. Static power is consumed when the transistors are idle and not actively switching between states. Subthreshold leakage current is a major contributor to the static power and its contribution to leakage power is increasing with decreasing feature size. Subthreshold leakage current is the current that flows from the source to the drain when the transistor is in the subthresold region, ie, $V_{GS} < V_T$. For devices with threshold voltage of 0.2V, the leakage current exceeds 50% of the power consumption. Traditionally, dynamic power has been the major contributor to the total power consumed in the chip and it depends on the square of the supply voltage.

$$P_{dynamic} = C*V^2*f$$

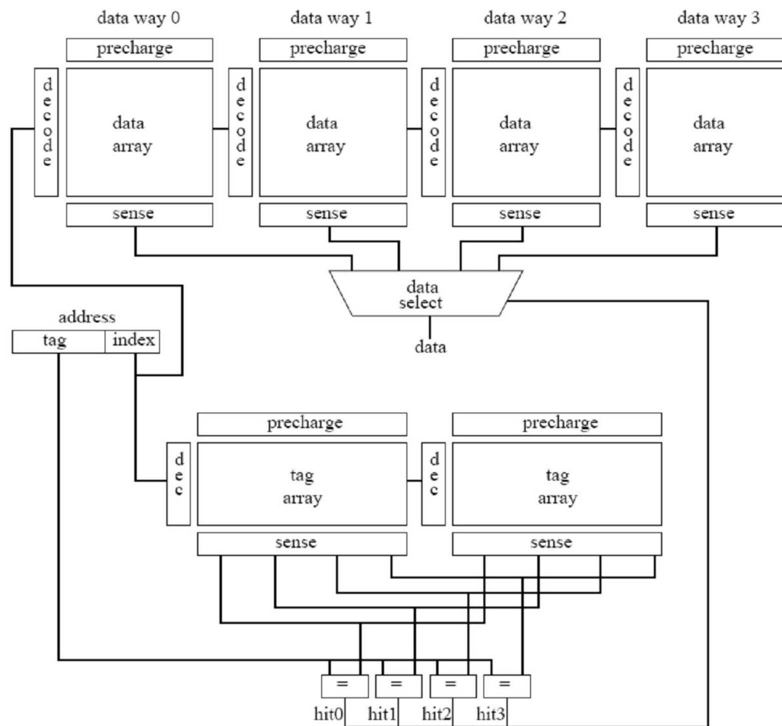


Fig 3.1: Way shutdown cache [1]

With the decreasing feature size, the supply voltage was decreased to reduce the dynamic power and also to reduce the strength of the fields within the device. But with reduced supply voltage the gate voltage also had to be decreased, which allows for only a small voltage swing below the threshold to turn the device off. The subthreshold conduction is exponentially related to the gate voltage and hence it is becoming more and more important as the devises are shrinking in size

Several techniques have been proposed to reduce the dynamic and static power consumed in on-chip caches. In [1] cache ways are selectively enabled or disabled by setting appropriate bits in the *cache way select register* (CWSR). A four way associative cache will have 4 bit CWSR as shown in Fig 3.1. The bits in CWSR signal if a particular way should be enabled or not according to which the cache controller enables or disables the way. The pre-charge signal to the cache ways is

gated such that ways that are not selected are not pre-charged. Since pre-charging bit lines is a major contributor to the dynamic power, disabling ways reduces the dynamic power consumption.

The increased power consumption in multiple way caches is due to the multiple parallel tag comparisons that occur for each cache look up. The way concatenation cache [19], as the name suggests, provides hardware support to concatenate ways of the cache into a single way thereby maintaining the cache size but reducing the number of ways. For many applications increasing the ways of the cache does not provide much performance improvement. In such cases concatenating the ways of the cache into a direct mapped cache helps reduce the power consumption. As show in Fig 3.2, ways are concatenated by using two additional control bits, *reg0*, *reg1* and the least two significant bits of the tag portion. When *reg0 = 1* and *reg1=1,* the cache is a normal 4 way set associative cache. When the control bits are set to *reg0 = 0* and *reg1 = 0,* the cache is a direct mapped cache with a total cache size which is four times the way size. The least two significant tag portion bits determine which of the four ways will be selected. For the control configurations *reg0 = 0* and *reg1 = 1* or *reg0 = 1* and *reg1 = 0,* the cache is a 2 ways set associate cache and only two ways are accessed. Again the total cache size remains the same and the way size is hence effectively doubled. Way concatenation ensures that a lesser number of ways are accessed whenever possible and hence reduces the dynamic power. In contrast to way shutdown caches, way concatenation maintains the same cache size and hence does not cause much degradation in performance.

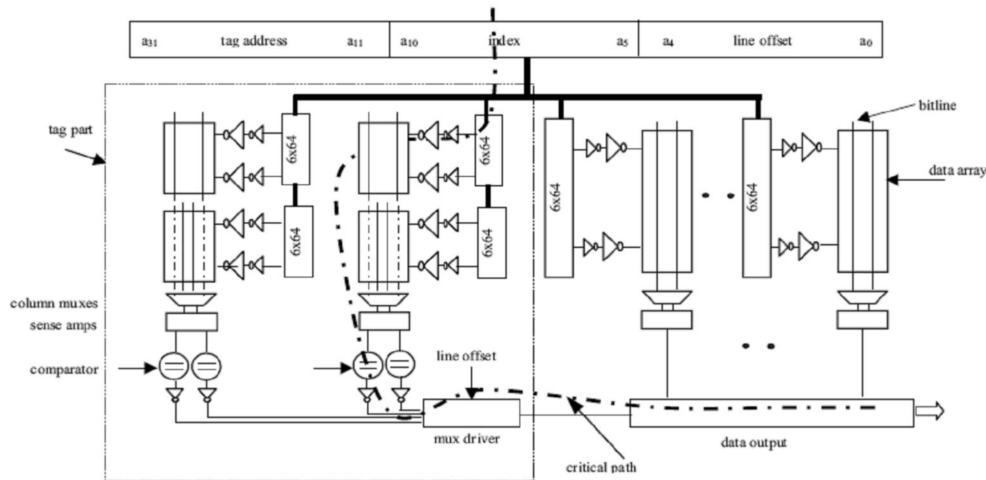


Fig 3.2: Way concatenation cache [19]

The techniques discussed above help decrease the dynamic power by controlling the number of ways that are accessed. Even when a certain way is not accessed there is still a certain leakage current flowing through the transistors contributing to the static power. To decrease the static power, the gated-V_{dd} technique introduces an additional transistor between the SRAM and ground reducing the leakage current by the stacking effect of transistors in series. Gated-V_{dd} though effective in decreasing the leakage current does not retain the data in the particular cache locations. As a result when these cache locations are accessed later, it results in cold misses and data has to be fetched from the lower memory levels. Alternatively, if these cache locations are maintained in a low power state instead of completely switching them off, the data can be retained along with decreasing the static power. This is the idea behind drowsy caches [3]. Through the use of Dynamic Voltage Scaling (DVS) technique, the cache lines which are to be put into the drowsy state are switched a low supply voltage.

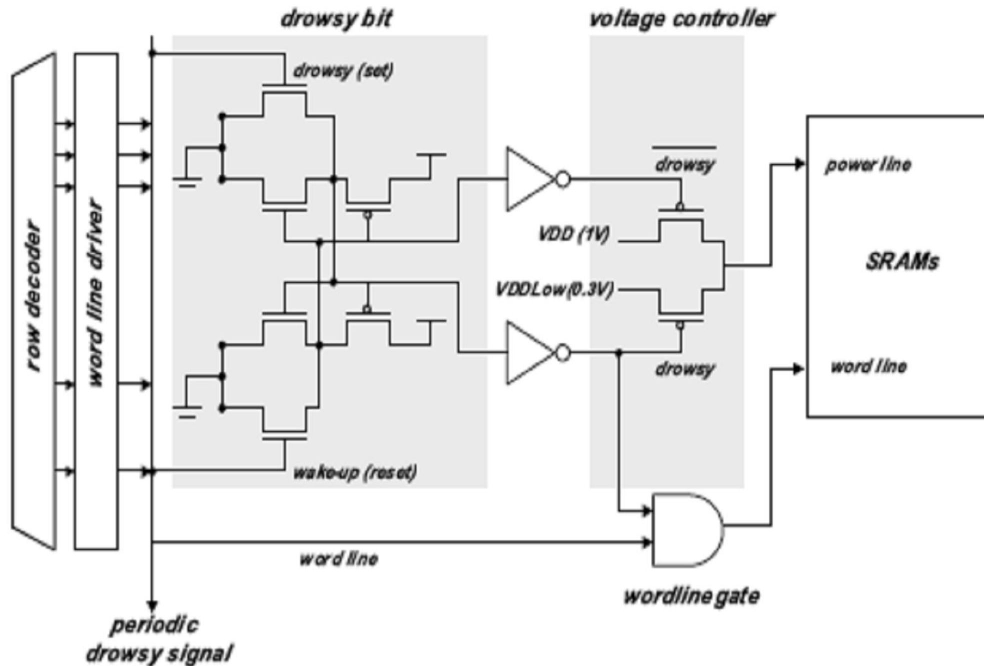


Fig 3.3: Drowsy cache line circuitry [3]

Due to short-channel effects in deep submicron processes, leakage current reduces significantly with voltage scaling [18]. The drowsy bit in Fig 3.3, determines whether a cache line is in the active state or drowsy mode and accordingly drives the cache line by high (active) or low (drowsy) supply voltages. The wordline is gated with the drowsy bit ensuring that drowsy cache lines are not read thus avoiding reading out wrong data. Whenever a drowsy cache line is accessed, the drowsy bit is reset, allowing the cache line to be read with a penalty of one extra cycle. This penalty is negligible compared to the several cycles of memory look up in the case of gated-V_{dd} because the data has to be reloaded. Hence in drowsy cache implementation, the cache lines can be aggressively put into the drowsy state with much degradation in performance.

Chapter 4: Cache Partitioning

4.1 The Benchmarks

We use video, image and speech encoding/decoding applications from the MediaBench suite [24] to define our benchmarks. Specifically we make use of the following MediaBench applications.

JPEG : Lossy image compression method for full-color and gray scale images.

MPEG2 : Encoder and decoder for MPEG-1 and MPEG-2 video bit-streams.

EPIC : Lossy image compression technique using critically-sampled non-orthogonal dyadic wavelet decomposition and a combined run-length/Huffman entropy encoder.

ADPCM : Adaptive Differential Pulse Code Modulation is a family of speech compression and decompression algorithms which converts 16 bit linear PCM samples to 4 bit samples thus giving a compression ratio of 4:1.

GSM : Full-rate speech transcoding algorithm using residual pulse excitation/long term prediction.

G.721 : CCITT (International Telegraph and Telephone Consultative Committee) G.721 speech coding specification.

In addition to the above mentioned MediaBench applications, we also make use of LAME, an MPEG Audio Layer III (MP3) encoder, in defining our benchmarks. MP3 players are common these days in many embedded systems like

cell phone, PDA and portable media player which prompted the inclusion of LAME into the set of applications under consideration.

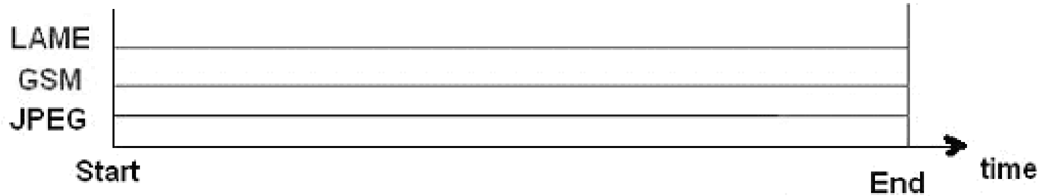


Fig 4.1: Timeline of a static benchmark

To evaluate our partitioning algorithm we defined benchmarks consisting of two or more applications. The benchmarks are categorized into two types based on the dynamicity of the start and end times of each task. A static benchmark is one where all tasks in the benchmark start and end at the same time. Fig 4.1 shows the timeline of a static benchmark containing 3 tasks. Typically the benchmark has a life time equal to that of the longest task. The load on other tasks is adjusted such that all tasks end at the same time. A dynamic benchmark on the other hand allows different tasks to start and stop at different times. Each task in the benchmark runs to completion and exits irrespective of whether other tasks are active or not. The benchmark is completely executed when all tasks complete execution once. At different time instances, a dynamic benchmark has different number of tasks active, whereas a static benchmark has the same number of tasks active from the start to end. Fig 4.2 shows a time line of a dynamic benchmark with 4 tasks.

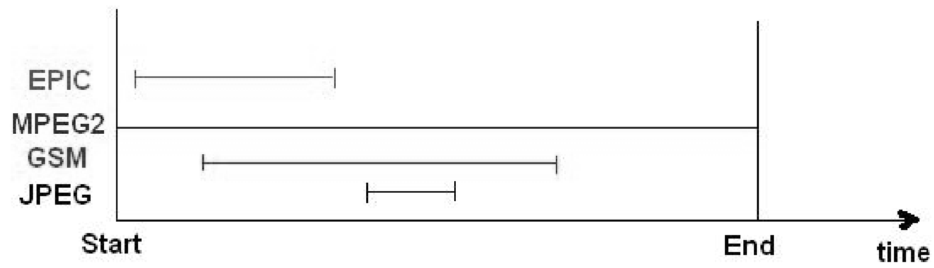


Fig 4.2: Timeline of a dynamic benchmark

The multimedia applications that constitute each benchmark used in our simulations are described in Table 4.1 and 4.2.

Benchmarks	Application1	Application2	Application3	Application4	Application5
BM1	MPEG2	LAME	G721		
BM2	JPEG	LAME	MPEG2	EPIC	
BM3	G721	LAME	GSM		
BM4	G721	MPEG2	JPEG	LAME	
BM5	G721	MPEG2	GSM	JPEG	
BM6	G721	LAME	JPEG	EPIC	ADPCM

Table 4.1: Dynamic Benchmarks

Benchmarks	Application1	Application2	Application3	Application4	Application5
SBM1	LAME	JPEG	G721		
SBM2	MPEG2	LAME	GSM		
SBM3	MPEG2	LAME	G721	EPIC	GSM

Table 4.2: Static Benchmarks

4.2 Partition Methodology

As mentioned earlier for most multimedia applications the miss rate settles after a certain cache configuration. Fig 4.3 shows the miss rate plots of the multimedia applications we have included in our simulations, for various cache configurations. Beyond a certain point, increasing the cache size or associativity ways provides only minimal gain in performance. We refer to this cache configuration as the point of minimal gain or the optimum cache configuration for that task.

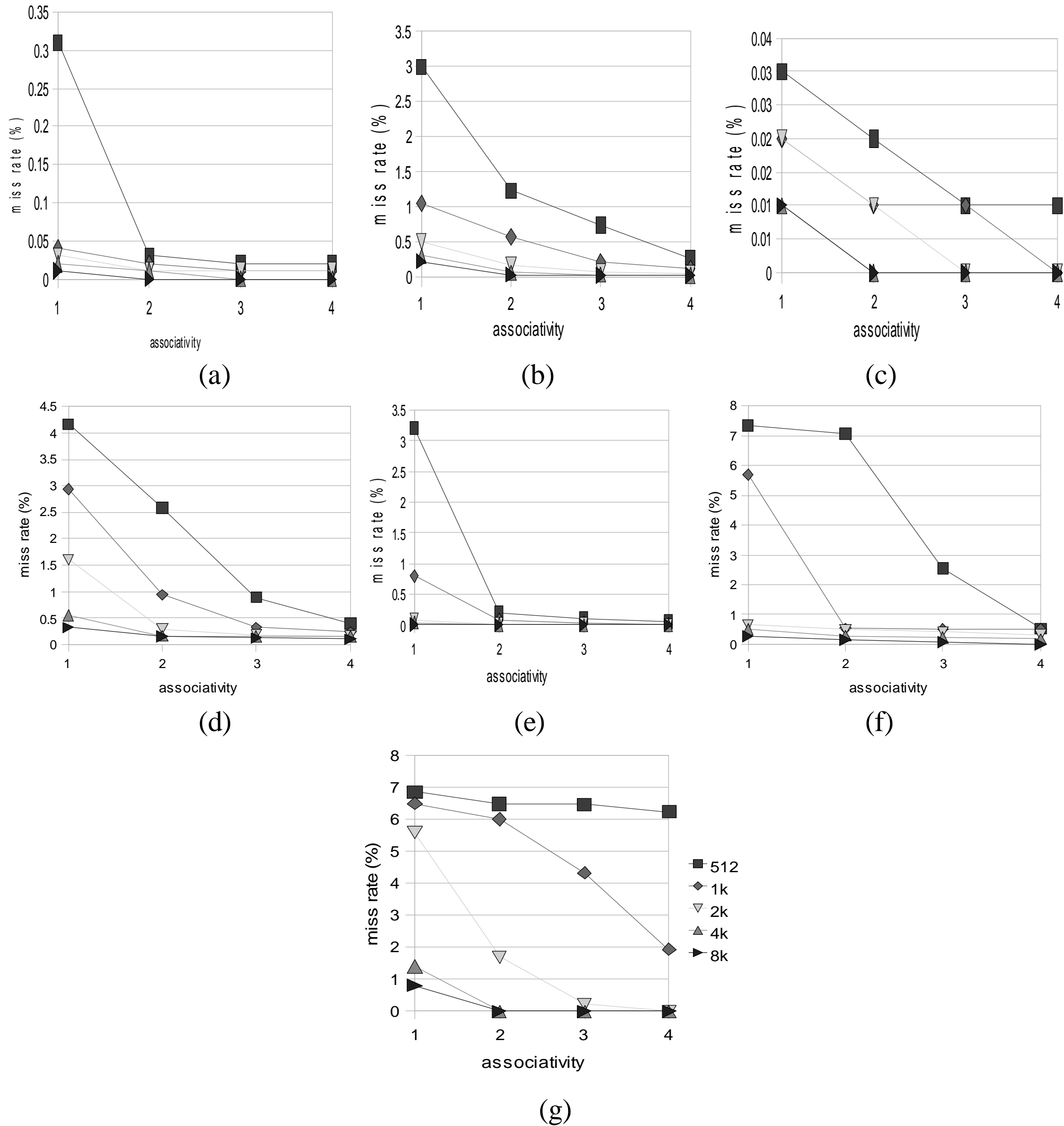


Fig 4.3: Miss rates a)EPIC b)JPEG c)ADPCM d) LAME e)MPEG2 f)GSM g)G721

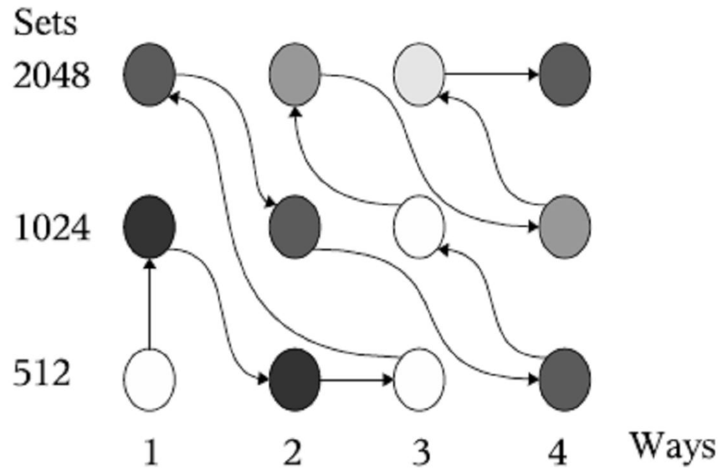


Fig 4.4: Exploration of partition space [2]

The cache partitioning starts with determining an optimum cache partition for each task. This cache configuration is determined based on the profiling information available at compile time, and the details of the cache configuration are included within the cache binary. The task binary also contains within it the miss statistics for different cache configurations in a table form. To determine the optimum cache configuration for each task, which we also refer to as the *starting configuration*, we compare the miss statistics of the task with a performance threshold and pick the first cache configuration, while following the cache space exploration pattern shown in Fig 4.4, with sub threshold miss rate. This exploration space considers different cache configurations in the increasing order of energy consumption. For a particular cache size we always consider the directed map cache first and then move on to caches with increasing number of cache ways in that order. The partitioning scheme tries to achieve performance close to that in the case where the tasks share the baseline cache.

Hence, if BASE(T_i) represents the miss rate for task T_i when it share the baseline cache with other tasks in the benchmark, the partitioning scheme aspires to achieve performance close to BASE(T_i). But in reality, BASE(T_i) is not available until the entire benchmark completes execution and hence is not available to be used as a threshold. Also, BASE(T_i) depends on the other tasks within the benchmark about which no information is known for each individual task. The partitioning scheme instead makes use of information local to each task and use IND_BASE(T_i), which is the miss rate of task T_i when using the baseline cache in isolation, to define a suitable threshold. Due to the phenomenon of cache interference,

$$\text{IND_ BASE}(T_i) <= \text{BASE}(T_i).$$

Using a tolerance value, k, to account for the degradation in performance due to cache interference, the threshold is defined as, (k + IND_BASE(T_i)). Based on experiments with different combination of tasks, we choose a value k = 0.1, indicating that we expect cache interference to cause an absolute increase of 0.1% on the average in the miss rate of each task. Using k = 0.1 is actually being conservative, because in actuality for many tasks cache interference causes much serious degradation in performance. But there are also tasks which are robust to cache interference effects and hence do not show degradations of the order represented by a value of 0.1 for k.

If MISS($P_{i,j}$) represents the miss rate of task T_i for a partition P_j, the starting configuration for each task is determined by picking the first cache configuration satisfying,

$$\text{MISS}(P_{i,j}) <= k + \text{IND_BASE}(T_i).$$

The partitioning problem now boils down to allocating the starting cache partition to each task as and when it is loaded into context. The fact that the determination of each task's cache partition is made using information local to that task alone is central to the run-time nature of the partitioning scheme. When a certain task begin execution the operation system, analyses the task binary to look up its starting configuration. Depending on where cache space is available to accommodate the cache partition, the OS sets the CWSR register [1] and mask register [15] to allocate the task its cache partition. At the time of context switch, along with saving other register values, the OS now also has to store the contents of CWSR and the mask register in order to allow the task to use the same partition when it is scheduled into context next. As and when a new task arrives, based on the starting cache configuration information present in its binary, the OS allocates the task a cache partition wherever it can be accommodated. Wherever possible the OS tries to avoid overlap between partitions. If a certain task's cache partition cannot be accommodated without overlapping it with some other task's cache partition, the OS determines from among the currently executing tasks, the one which exhibits maximum robustness towards cache interference, and tries to allocate the new cache partition such that it overlaps with the partition of the most robust task. This is done to minimize the effects of cache interference due to the overlapping partitions. The OS can determine the most robust task by analyzing the miss statistics table of the tasks and determining the slope of the miss rate around the current partition. A sudden drop in miss rate just before current partition indicates that the task may be able to accommodate a certain amount of overlap with its current partition. Counters keep track of the miss rate for the tasks

with overlapped partitions and check if they are within the threshold. If not the overlapped partitions are subject to and update policy explained later.

Consider a benchmark with three tasks.

Task 1: G721

Task 2: GSM

Task 3: LAME

For k=1.1, the starting configurations of G721, GSM and LAME are 8kB,2way, 8kB,2way and 4kB,4way respectively. Fig 4.5 (a) depicts the layout of the partitions of individual tasks after the partitioning process. LAME partition cannot be allocated exclusive of the other partitions and hence has to be overlapped with that of G721 and GSM.

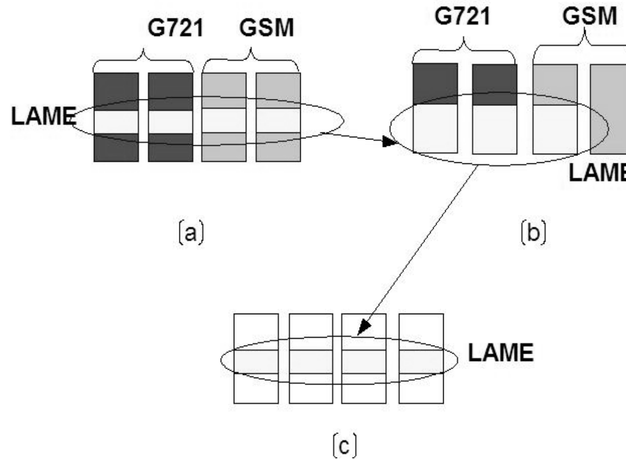


Fig 4.5: (a) Cache partitioning for a 3 task benchmark. (b) Effect of partition update. (c)LAME partition after partition reshuffling.

When a certain task is being executed, only its partition is kept active and that of other applications are put in a low power drowsy mode [3]. Having the inactive cache partitions in drowsy mode helps save leakage power as explained in [3]. In addition to leakage power savings, the partitioning scheme also gives savings in dynamic power.

As mentioned earlier, dynamic power is due to the charging and discharging of the various capacitive loads at the switching time. Precharging the bit lines is one of the major contributors towards this. As explained in the architectural modifications suggested in [1], the precharge signal being gated with the *enable_ways* signal, only the active partitions need to be precharged. This reduction in the capacitive load, helps reduce the dynamic power.

4.3 <u>Partition Update</u>

It was previously mentioned that overlapped partitions will be required to undergo partition update whenever they exceed performance thresholds. Overlapped partitions suffer from the problem of cache interference, as was in the case of baseline cache. Hence the actual miss rates for these tasks could be worse than MISS($P_{i,j}$), which is the miss rate for the task T_i when running in isolation with a cache of size $P_{i,j}$. In an effort to provide for performance within the defined performance bounds, power savings is traded off for improved performance. The miss rates of the tasks with overlapped partitions are monitored regularly and when the miss rate exceeds the threshold value, the partition is updated to once which is expected to give a lesser miss rate. This update is based on the miss statistics provided in the task binary. The update of a cache partition may involve increasing either the way size, or the number of ways, or both, depending on the particular task and the current cache partition that is allocated to it.

When a task's partition needs to be updated, the OS looks up the cache miss statistics table and find the next cache configuration in the order shown in Fig 4.4, which satisfies the threshold condition mentioned above. After the update process is

completed, the miss-rate is allowed to settle down, by letting the tasks use the updated cache partitions for a certain period of time. This usually corresponds to a few context switches. The OS, at the time of context switch, checks the gradient of the miss rate by maintaining a moving window of the miss rate of the current task and indicates the miss rate to have settled once the gradient is suitably flat. If the miss rate still exceeds the performance bound, the partition is subjected to further update procedures as before. The update procedure can be carried on till the entire baseline cache is used by the task. The more the number of partitions updated and the more the number of times each partition is updated, the closer we move to the baseline situation. Hence making tradeoff between performance and energy savings, we stop updating a particular partition after it has been updated four times. The performance of a task whose miss rate hasn't yet come below the threshold would still be close to the threshold due to the update procedure. If during the update process any new partitions are overlapped, then their miss-rates are also tracked and each of these partitions subjected to the update procedure in case their miss rate exceeds the threshold.

Fig 4.5 (b) shows the effect of the partition update procedure on the benchmark with three tasks. The LAME partition overlaps with both GSM and G721 partitions and hence all three task's miss rates are tracked regularly and compared with their individual thresholds. GSM requires most of the 8kB, 2way partition allocated to it and is very sensitive to cache interference. As a result its miss rate exceeds the performance thresholds and the partition is subjected to the update procedure. GSM partition is updated to 12k,3way partition and LAME which also exceeds the threshold is updated to 6k,3way cache partition. G721, on the other hand,

is less vulnerable to cache interference and continues to give sub-threshold miss rates. The distribution of the cache partitions after update procedure is shown in Fig 4.5 (b). Fig 4.6 gives a heuristic partition update algorithm which the OS has to perform at regular intervals. Here *COST(P)* refers to the cache space consumed by the partition of the task under consideration and *COST(cache)* is the total available cache space. If the baseline cache is 16kB, 4way, then *COST(cache)* is 16kB. The update procedure results in each application having a bigger partition which would help decrease the miss rates and eventually bring it below the threshold. Since the partitions are grown, the scope for energy reductions are reduced by decreasing the area that can be put into low power mode and also increasing the number of cache lines that need to be precharged. In spite of this, the cache partitions are still smaller than the baseline cache and hence continue to provide energy savings over the baseline case.

```
//initialize the update count for each partition to zero.
for all i
   partition_update_count[i] = 0

for all i
   P = Pi,j
   if(current_missrate(task i) > threshold[i])
      if(partition_update_count[i] < 4 and COST(P) < COST(cache))
         //update the current partition of task i
         P = next partition, Pi,k , along the exploration path with MISS(Pi,k) < threshold[i]
```

Fig 4.6: Heuristic partition update algorithm

4.4 Partition reshuffle

When a task completes, its partition is freed and is available to be allocated to other executing tasks. We define partition reshuffling as the process of relocating a cache partition from an overlapped region to a non overlapped one when previously

occupied cache has been freed by a completing task. If the partition that is being relocated is one that has already been updated, then the reshuffled partition size is set to the starting configuration for that task. The rationale for this being that, when no overlap exists between partitions, the starting configuration guarantees sub threshold performance. Also moving to a smaller cache partition, implies more power savings allowing to compensate for the power savings lost during the update procedure. If more than one partition has been updated and currently overlap, we have more than one contender for the reshuffling process. We use a simple arbitration scheme which considers the applications in the scheduling order. The first application, among the contending applications, to be scheduled into context by the OS is considered first for the reshuffling process. If after the reshuffling, there still remains enough space, the next application in the scheduling order whose starting configuration can be allocated in the available space is relocated.

In Fig 4.5 (b), after the partition update procedure there is still overlap between all three tasks. As part of the update procedure the LAME partition was updated to 6k,3way set associative cache. If GSM and G721 complete before LAME, there is no further overlap of the LAME partition with any others. Hence the starting cache configuration would suffice to maintain the sub-threshold miss rate for LAME. LAME is therefore allocated a 4k,4way cache partition as part of the reshuffling process and this partition can be relocated to any valid position within the baseline cache. One possible positioning of this cache partition after the reshuffling process is shown in Fig 4.5 (c).

Another scenario where a task whose partition has not been updated is subjected to partition reshuffling is shown in Fig 4.7. The tasks sharing the cache are GSM, G721 and MPEG2. GSM and G721 require 8k, 2way cache partitions which are respectively represented by the distinctly colored set of two rectangles each. The 2k, 1way cache partition of MPEG2, due to lack of cache availability, has to be overlapped with either of the previous two partitions. G721 being more robust to cache interference, the MPEG2 partition is overlapped with that of G721. In the scenario where GSM completes and exits before either of the other two tasks complete, we can make use of the 8k,2way cache freed by GSM to eliminate the cache partition overlap. This partition reshuffling or relocation results in the layout shown to the right of the Fig 4.7. In this new partition layout, there is no overlap between the partitions of G721 and MPEG2 and hence both yield better performance. In this reshuffling example, there is no shrinking of the partitions, instead only a relocation of a currently overlapped partition to a non overlapped one. This allows for better performance. The arbitrator while checking the tasks with overlapped partitions, gives first priority to updated partitions and only when no other updated partitions can be relocated, allow relocations like the ones mentioned in Fig 4.7

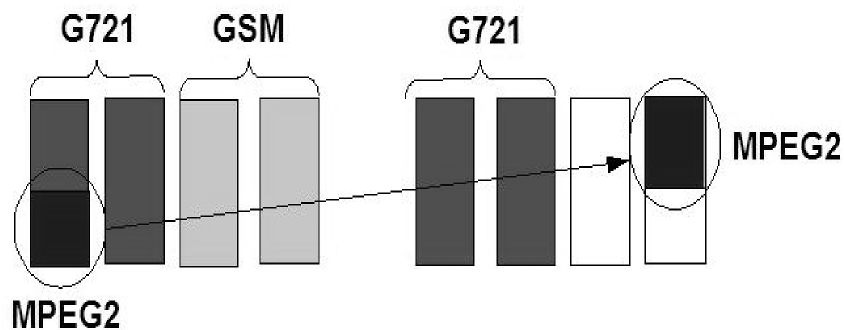


Fig 4.7: Example of cache partition reshuffling

Chapter 5: Evaluation environment

5.1 The Simulator

DineroIV[26], a trace driven simulator, only allows a single cache to be simulated. To study our cache partitioning algorithm, multiple cache partitions, each possibly of different size and associativity ways, need to be simulated. In our simulator, we simulate caches of any size and associativity way by constructing larger caches using smaller direct mapped caches as the building blocks. Individual 512 byte caches are simulated, using a framework similar to the dineroIV simulator and are used as the building blocks. Since the largest cache configuration we use in our simulations is a 32kB cache with a maximum of 4 ways, 64 separate 512 byte caches are required and these are arranged in a 16 X 4 grid as shown in Fig 5.1.

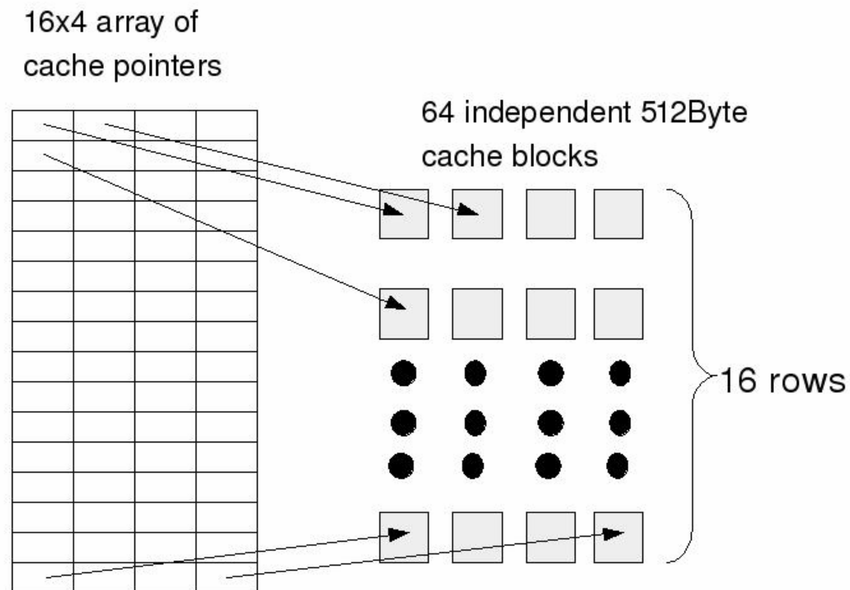
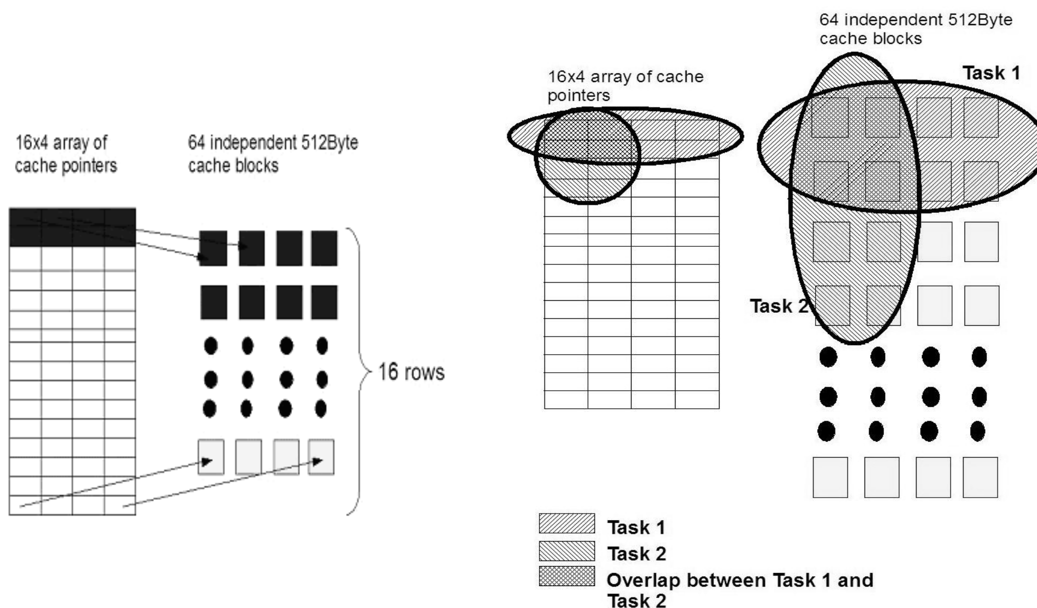


Fig 5.1: 32kB, 4way cache modeled using 64 independent 512 byte caches.

To allow separate tasks use different sections of the cache we maintain a 16 X 4 array of pointers with each entry in the array pointing to one cache instance in the 16 X 4 grid. If for instance a 4kB, 4way cache partition is to be allocated to a certain task, this task is allocated two rows of cache pointers depending on the desired location of the partition in the grid as depicted in Fig 5.2 (a). Each application accesses its partition through its set of cache pointers. Partition can be allowed to overlap by assigning the task in consideration a common set of cache pointers, which means that they would use the same cache instance and interfere with each other. Fig 5.2 (b) shows an example of how a 4kb, 4way cache partition allocated to Task1 can be overlapped with a 4kB, 2way partition allocated to Task2, by making the tasks access cache via the appropriate set of cache pointers.



(a) *(b)*

Fig 5.2: (a) Allocation of a 4kB, 4way cache partition, (b) overlapping a 4k, 4way partition with a 4k, 2way partition

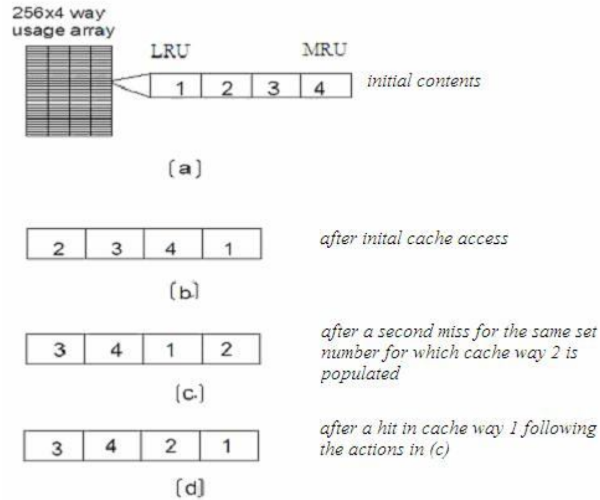


Fig 5.3: Way usage record for modeling multiple cache ways.

To simulate caches with multiple cache ways, additional book-keeping structures are required to keep track of the order in which the ways are accessed and to determine which cache way will be replaced in case of a miss in all the ways. A *way usage* record is hence maintained which has an entry for each set in the cache. For a four way set associative cache, the record has four entries for each set indicating the relative order in which the cache ways were accessed. Fig 5.3 (a) gives a detailed view of the *way usage* record for one set number with its initial values. The rightmost entry in each field indicates the most recently used cache way and the left most entry the least recently used way. We use the Least Recently Used (LRU) replacement algorithm in our cache simulations, hence this left most cache way is the candidate for replacement. After each replacement or hit in the cache way, the *way usage* record is appropriately updated for that set number. Fig 5.3 (b) to (d) show how the contents of the *way usage record* are update for successive accesses allowing easy simulation of the LRU replacement algorithm.

Fig 5.4: Block diagram representation of complete experimental setup

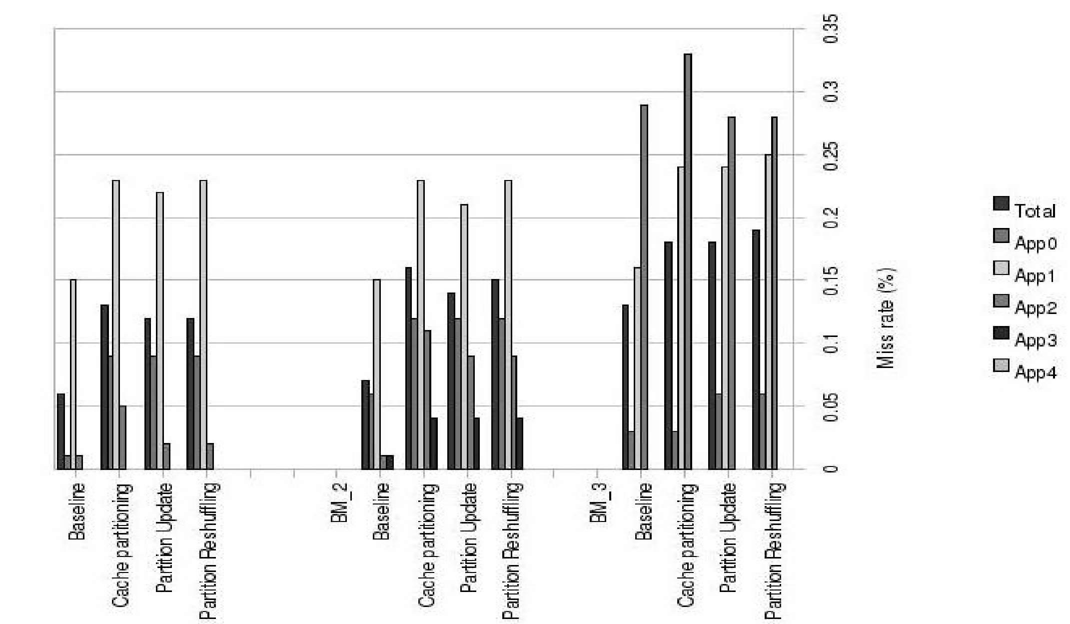
A block diagram representation of our entire evaluation environment is shown in Fig 5.4. We use sim-cache tool from the simplescalar toolset [17] to generate the traces of each of the applications mentioned in the section on benchmarks. The unused most significant bits in the address space of each application is used to define a unique process identifier (PID) to each application. The traces of the tasks that constitute a particular benchmark, as described in Tables 3.1 and 3.2, are fed as input to our simulator which gives us the miss statistics for each task for the partitions allocated to it. These miss statistics used as inputs to the CACTI tool for calculating the energy consumption values.

Chapter 6: Experimental Results

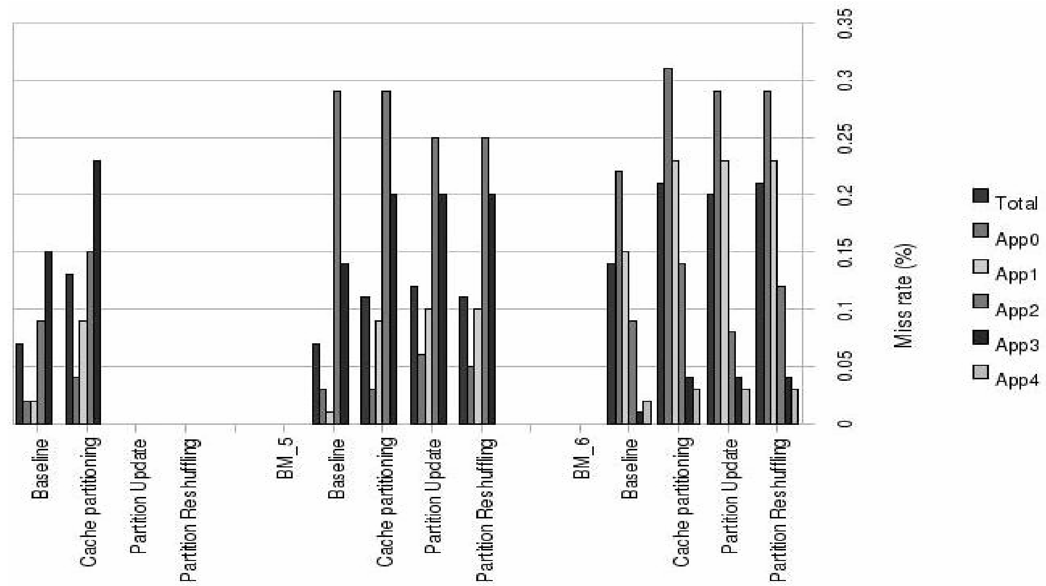
We evaluated the partitioning algorithm for both dynamic and static benchmarks for 16kB, 4way and 32kB, 4way set associative baseline caches for context switching frequency of 33k instructions. Cache configurations ranging from 512kB to 16kB with associativity ways from 1 to 4 were considered. While modeling cache partitions odd numbered associativity ways were also allowed. CACTI version 5.3 cache modeling tool was used to evaluate the dynamic and static energy consumption values.

6.1 Impact on performance

Fig 6.1 compares the miss rates of each task and the benchmark as a whole for each of the three partitioning schemes. When only partitioning is allowed, the performance of tasks tends to exceed performance levels in case of partition overlap. This is clearly visible in the case of benchmark BM_3 in fig 6.1 (a) where App2, which is GSM exceeds its threshold limit of 0.27%. The update procedure in this case helps keep the miss rates within the threshold and. Updating the GSM partition has a detrimental effect on the performance of LAME, which is also updated in the process to keep it's performance within it's threshold value of 0.25. Partition reshuffling primarily targets energy savings, and decreases the partition budget whenever free cache is available and an overlap can be avoided. As can be seen from Fig 6.1, since reshuffling allocates the starting configuration to the concerned task, the miss rate continues to be retained within the threshold value.



a)



b)

Fig 6.1: Miss rate comparison for dynamic benchmarks for 16kB baseline cache

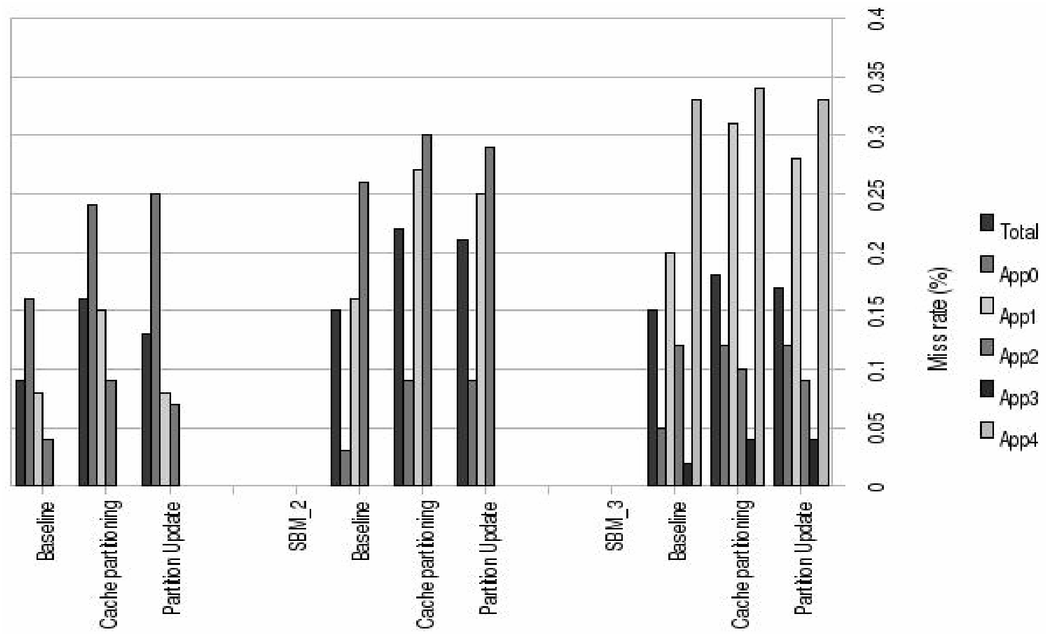


Fig 6.2: Miss rate comparison for static benchmarks for 16kB baseline cache

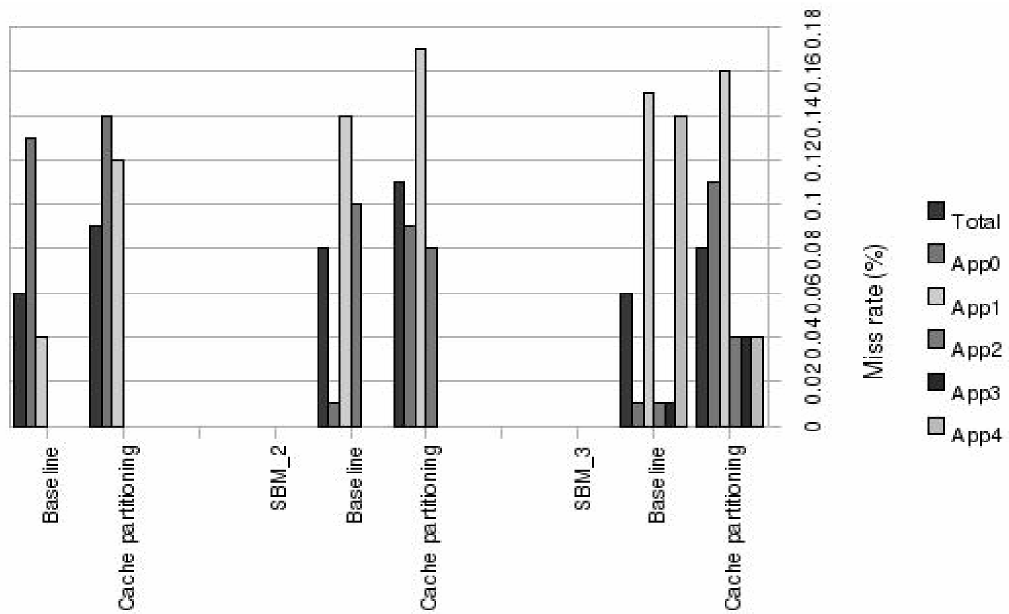
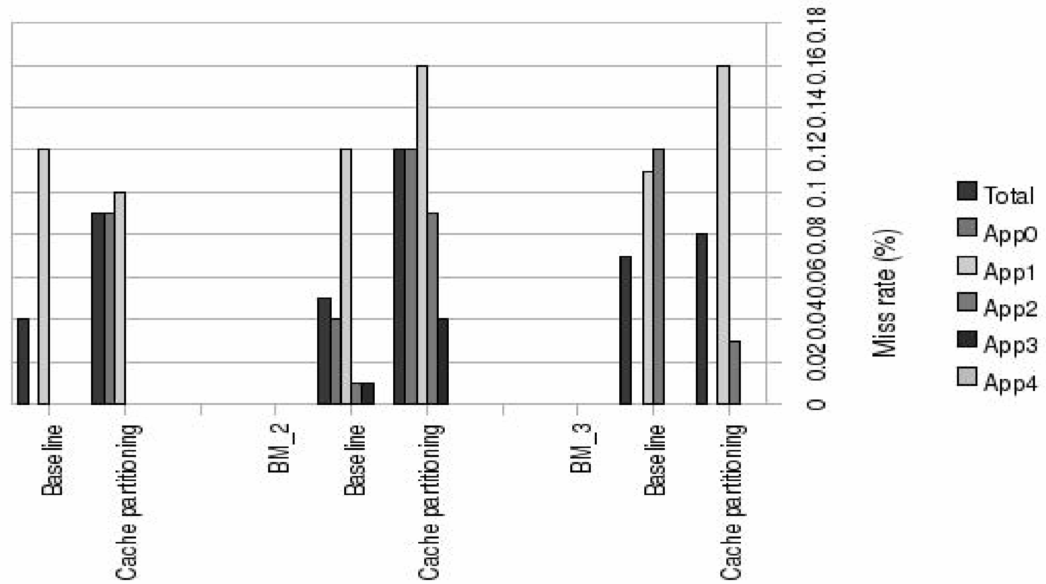
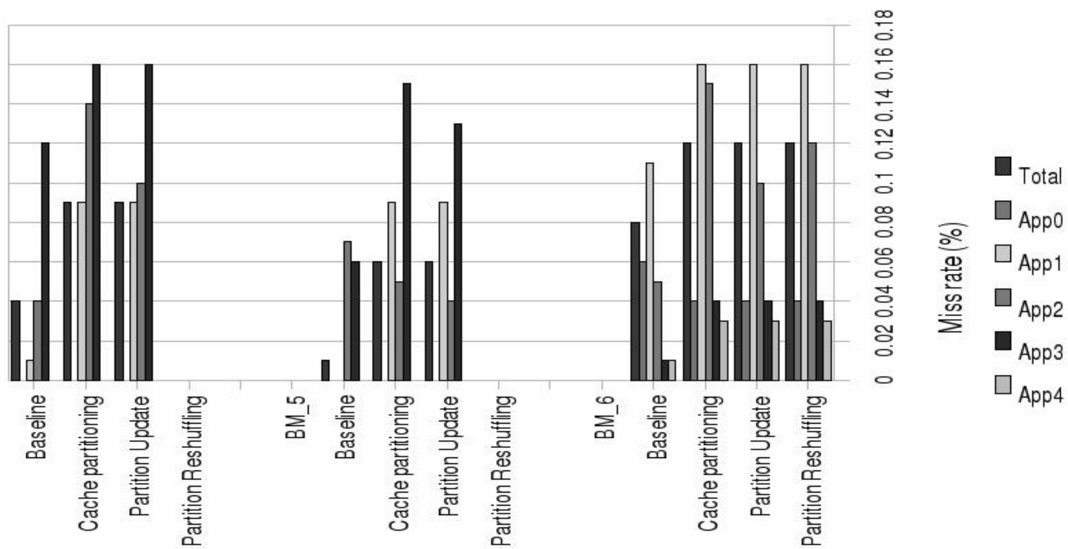


Fig 6.3 Miss rate comparison for static benchmarks for 32kB baseline cache



a)



b)

Fig 6.4: Miss rate comparison for dynamic benchmarks for 32kB baseline cache

Fig 6.3 and Fig 6.4 shows the impact of the three partitioning policies on the miss rates of the benchmarks for a 32kB, 4way set associative baseline cache. As can be seen, due to the larger cache space availability, overlap between partitions is rare and

hence cache partitioning by itself normally achieves the performance thresholds. In the case of BM_6 though, due to partitions overlapping with each other, the update procedure is called whenever the miss rate threshold is exceeded.

6.2 Impact on power

We evaluated the impact of the algorithm on both dynamic and leakage power. CACTI-5.3 [16] was used to compute the dynamic and leakage power values for the various cache configurations. Each cache partition was modeled as a separate cache and it's dynamic and leakage power values were computed using CACTI for the 70nm technology. In the computation of the dynamic energy, cache misses were modeled as accesses to a 1MB, direct mapped SRAM.

Fig 6.5 and 6.6 show the percentage reduction in dynamic energy consumption as a result of the cache partitioning policies, compared to the baseline cache where all applications share the entire cache. In most cases the cache partitioning algorithm is able to reduce the dynamic energy consumption by at least 50% and at least 35% reduction in the worst case. The update policy as seen earlier helps keep the miss rates within the performance thresholds but decreases the energy savings. In some of the benchmarks like BM_1 and BM_2, the partition update policy causes only a minor reduction in the energy savings. BM_3 is a typical example of a benchmark where energy savings is clearly traded for performance as can be seen from the nearly 10% reduction in energy savings compared the case with no update. The increased dynamic energy consumption due to the update policy is compensated to a small extent by the reshuffling policy. Figs 6.7 and 6.8 show the dynamic energy savings for the 32kB,4way set associative baseline cache. By following a partition allocation

policy which allocates partitions along the periphery far removed from each other, the larger cache budget prevent any overlap in most cases. In cases of partition overlap, the update policy with possible reshuffling helps keep the miss rates under check. Larger savings are seen for the 32kB baseline cache with savings usually being between 50% - 65%.

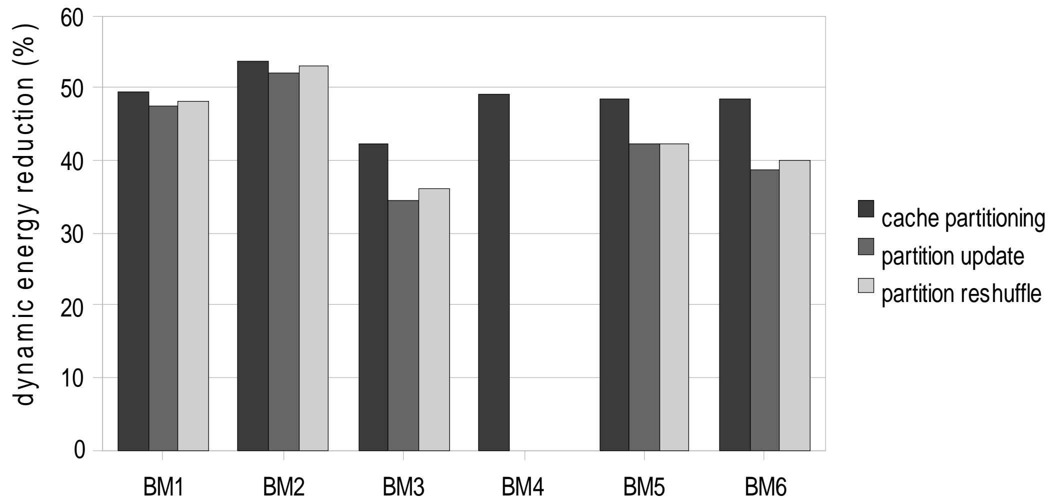


Fig 6.5: Percentage reduction in dynamic energy for dynamic benchmarks with 16kB baseline cache for 70 nm technology

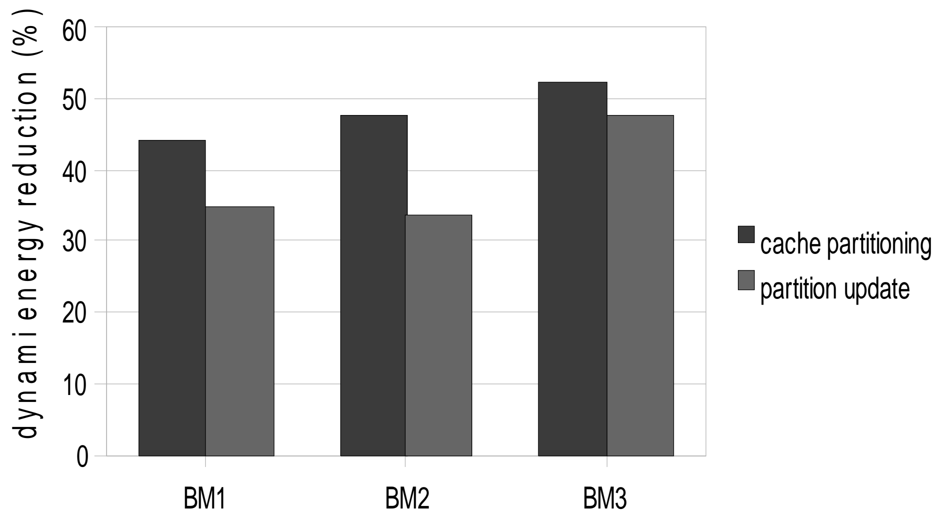


Fig 6.6: Percentage reduction in dynamic energy for static benchmarks with16kB baseline cache for 70 nm technology

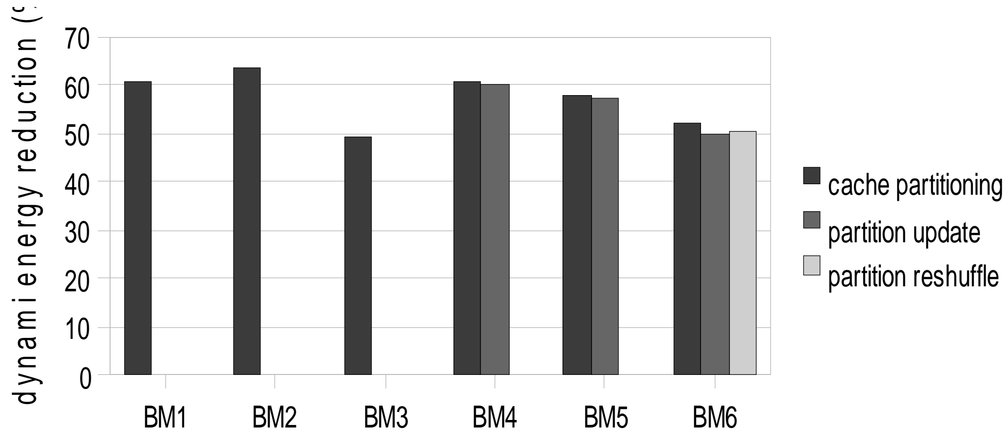


Fig 6.7: Percentage reduction in dynamic energy for dynamic benchmarks with 32kB baseline cache for 70 nm technology

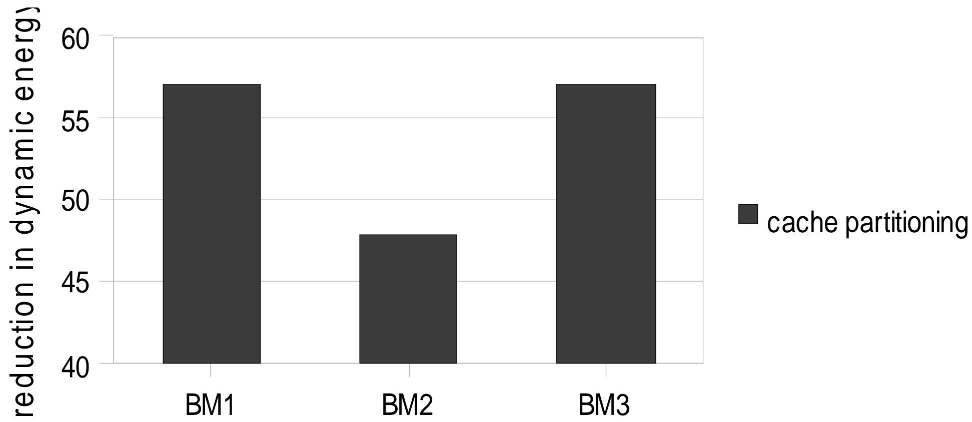
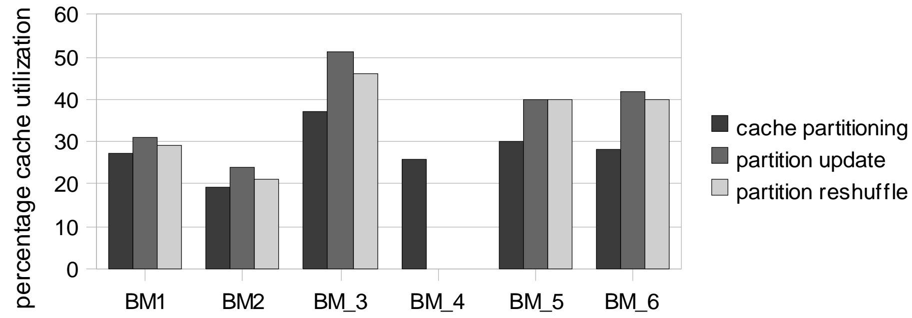
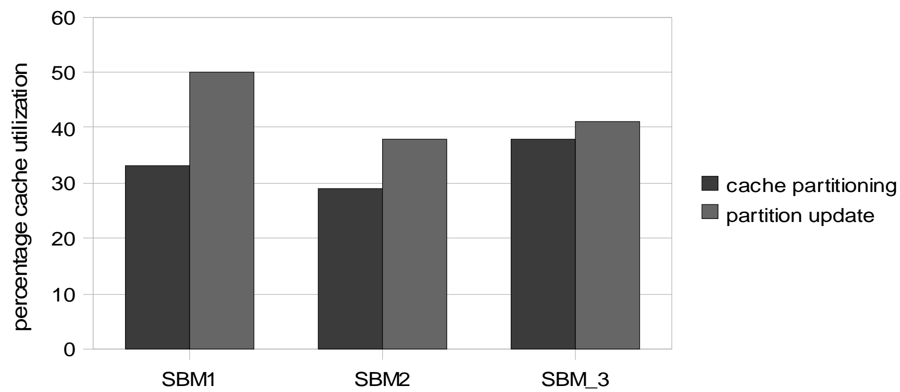


Fig 6.8: Percentage reduction in dynamic energy for static benchmarks with 32kB baseline cache for 70 nm technology

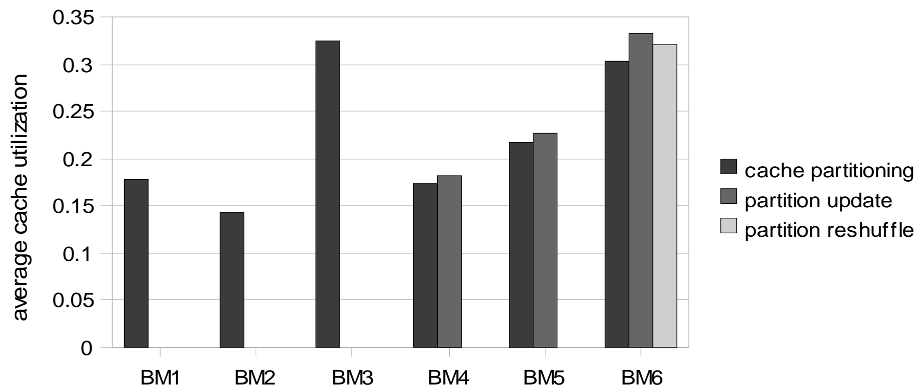
Since only a section of the entire cache is active at a certain instant, the remainder is put into a drowsy mode to save on the leakage power. Figs 6.9 and 6.10 show the savings in leakage power as a result of coupling the drowsy cache architecture [3] with our partitioning algorithm. In [3], the author reports the leakage power savings for the 0.18um technology, so to determine the actual savings possible for 70nm technology; we have referred to [27]. We first report the average cache utilization factor for each of the cache partitioning policies to show what factor of the cache is left unutilized, on the average, as a result of the partitioning scheme and hence available to be put into drowsy mode.



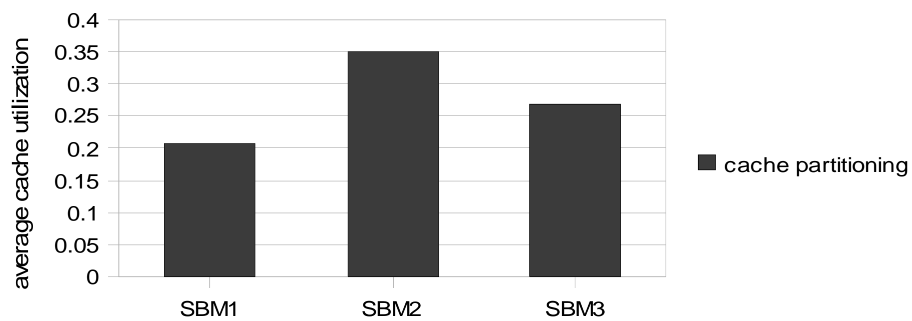
(a)



(b)



(c)



(d)

Fig 6.9: Average cache utilization: (a) dynamic benchmarks (b) static benchmarks with 16kB baseline cache; (c) dynamic benchmarks (d) static benchmarks with 32kB baseline cache

Fig 6.9 shows that maximum cache utilization on the average is only 50%, implying that more than half the cache can be put into low power mode to save on leakage power. We report the leakage reduction for 70nm technology (Fig 6.10 – Fig 6.13) and for 0.18um (Fig 6.14 – Fig 6.17).

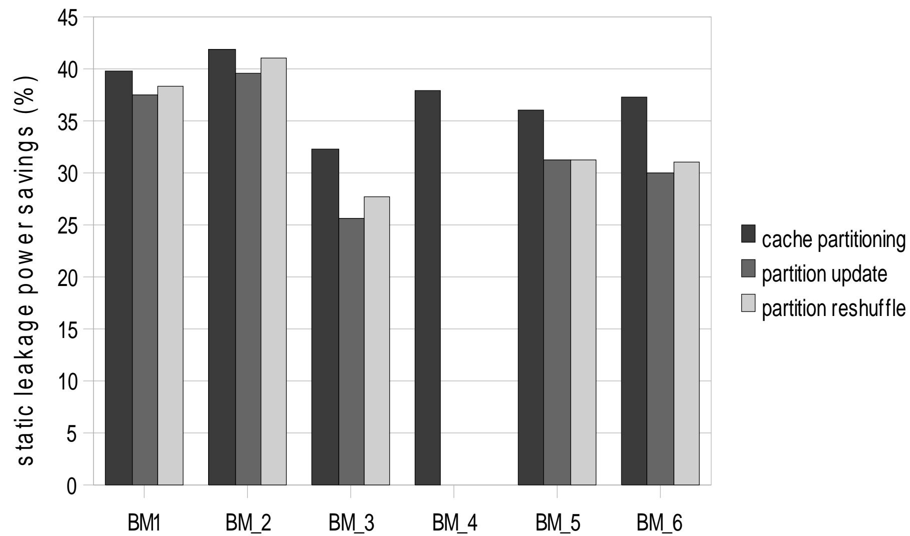


Fig 6.10: Percentage reduction in static leakage power for dynamic benchmarks with 16kB baseline cache for 70 nm technology

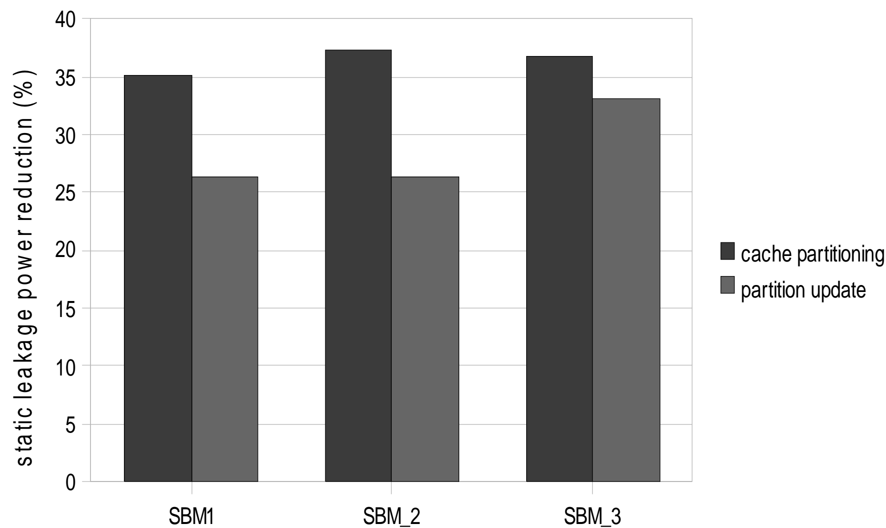


Fig 6.11: Percentage reduction in static leakage power for static benchmarks with 16kB baseline cache for 70nm technology

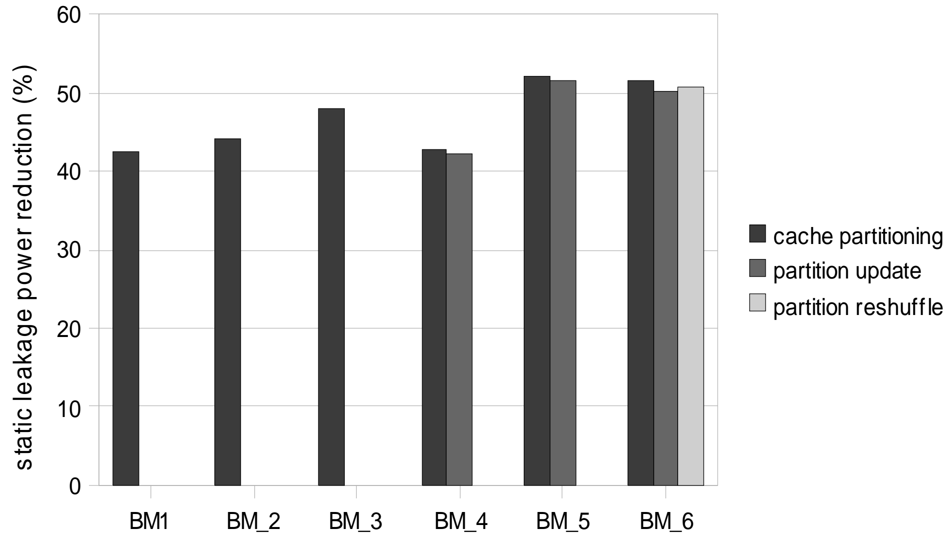


Fig 6.12: Percentage reduction in static leakage power for dynamic benchmarks with 32kB cache for 70 nm technology

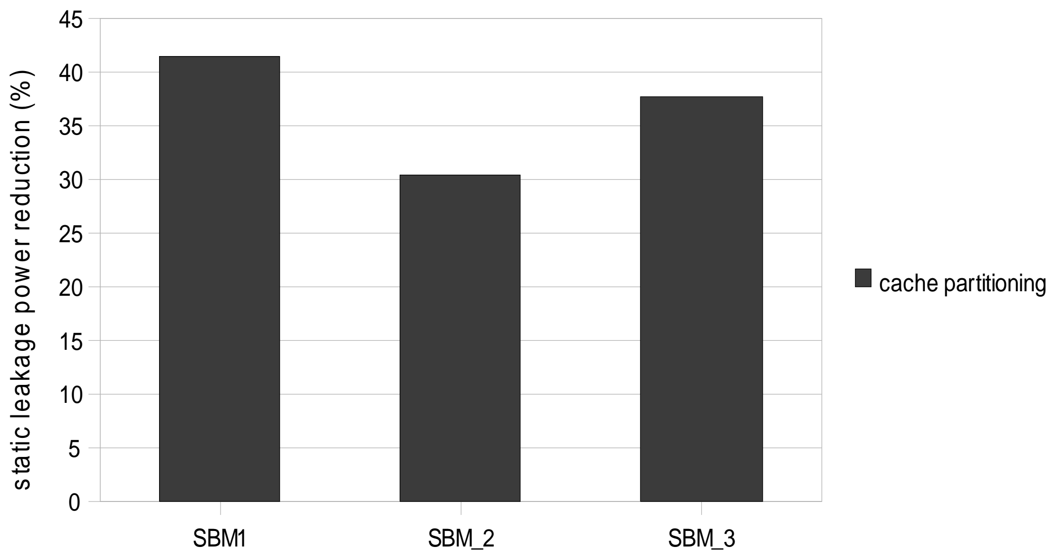


Fig 6.13: Percentage reduction in static leakage power for static benchmarks with 32kB baseline cache for 70 nm technology

Leakage power savings of the order of 30% - 50% is seen for our benchmarks. Static benchmarks exhibit larger reduction in static leakage power compared to the dynamic benchmarks. This is due to the fact that in static benchmarks applications like EPIC and ADPCM, which require much smaller cache partitions compared to applications like GSM and G721, use and equal proportion of the execution time of

the complete benchmark. Whereas in dynamic benchmarks if these applications run only once to completion, there execution time is much small compared to say G721 or LAME and hence the proportion of time for which the larger section of the cache is left in drowsy mode is less, resulting in the lesser reduction in static leakage power.

We also determined the static leakage power savings for the 0.18um technology for which the drowsy cache architecture [3] reports leakage power reduction by a factor of 12. Figs 6.14 to Fig 6.17 show the leakage power reductions obtained for the 0.18 um technology by coupling the drowsy cache architecture with our partitioning methodology. As mentioned in [3], for 0.18um technology the leakage power is reduced by a factor of 12 whereas for 70nm technology, as determined from [27], the reduction in only by a factor of 2.07. Even if the factor reduction is less, since leakage power is a greater contributor to the total power in sub-micron technology, the savings obtained by our approach are significant. Savings of the order 50%-80% are seen for the 0.18 um technology.

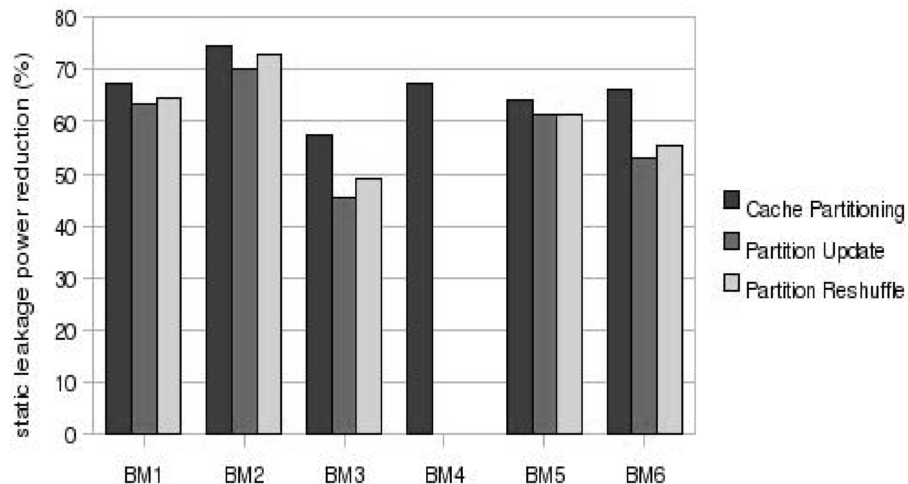


Fig 6.14: Percentage reduction in static leakage power for dynamic benchmarks with 16kB baseline cache for 0.18 um technology

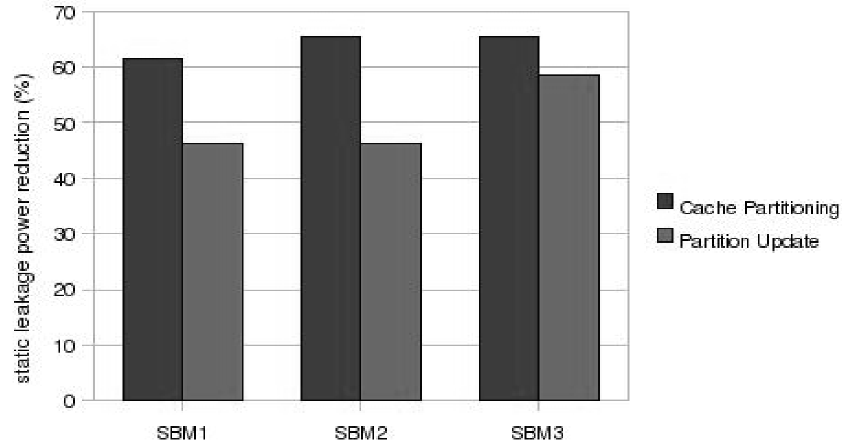


Fig 6.15: Percentage reduction in static leakage power for static benchmarks with 16kB baseline cache for 0.18 um technology

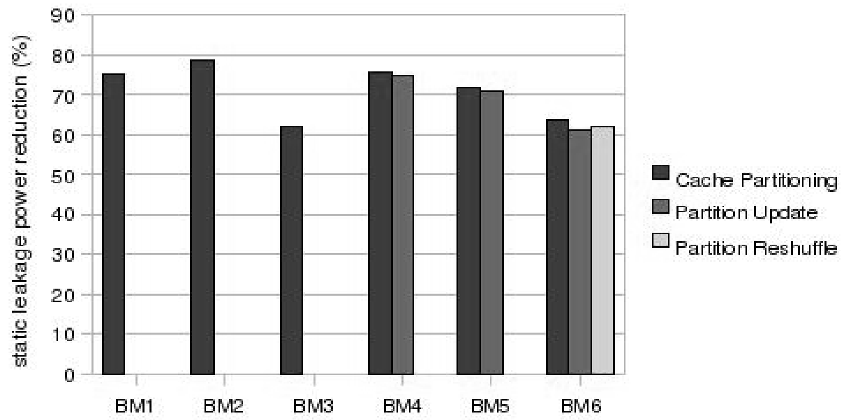


Fig 6.16: Percentage reduction in static leakage power for dynamic benchmarks with 32kB cache for 0.18 um technology

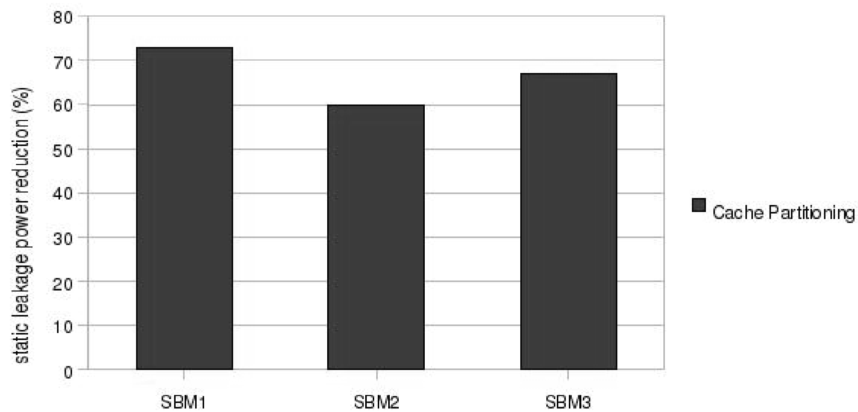


Fig 6.17: Percentage reduction in static leakage power for static benchmarks with 32kB baseline cache for 0.18 um technology

Chapter 7: Conclusion

We have proposed a methodology for partitioning at run time the instruction cache according to the requirements of each application. The work leverages recent work on reconfigurable caches that is available in the form of way-shut down and way concatenation caches and also the drowsy caches which allow selected portion of the cache to be put to a low power mode. By giving each application its own exclusive cache partition, the effect of inter-task cache interference can be eliminated and we can achieve close to the baseline cache performance by using smaller dedicated cache partitions. In cases of lesser cache availability than required by each application, the partitions need to be overlapped and the *partition update* and *partition reshuffling* policies ensure that the performance of applications are within their thresholds. When a certain application is active, the partitioning scheme allows other application partitions to be kept in a low power mode saving both dynamic and leakage power. Reductions in the order of 35%-65% are seen in dynamic energy with more being possible in larger caches. Static leakage power reductions of the order of the order of 30%-50% in 70nm and 50% - 80% in 0.18um technology are commonly observed by putting inactive partitions in the drowsy mode.

Bibliography

[1] D. H. Albonesi, “Selective Cache Ways: On-Demand Cache Resource Allocation”, in International Symposium on Microarchitecture (MICRO), pp. 248–259, November 1999.

[2] Rakesh Reddy, Peter Petrov, “Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems” in Compilers, Architectures and Synthesis for Embedded Systems (CASES), pp.198-207, 2007.

[3] K. Flautner, N. Kim, S. Martin, D. Blaauw and T. Mudge, “Drowsy caches: simple techniques for reducing leakage power”, in International Symposium on Computer Architecture (ISCA), pp. 148–157, May 2002.

[4] J. Mogul and A. Borg, “The effect of context switches on cache performance”, in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 75–84, 1991.

[5] D. B. Kirk, Jay K. Strosnider, “SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000”. in IEEE Real-Time Systems Symposium, pp.322-330, 1990.

[6] A. Wolfe, “Software-based cache partitioning for real-time applications”, Journal of Computer and Software Engineering, vol. 2, n. 3, pp. 315–327, 1994.

[7] F. Mueller, “Compiler support for software-based cache partitioning”, in Languages, Compilers, and Tools for Embedded Systems (LCTES), pp.125–133, 1995.

[8] K. Inoue, T. Ishihara, K. Murakami, "Way-predicting set-associative cache for high performance and low energy consumption" in International Symposium on Low Power Electronics and Design, pp. 273-275, 1999.

[9] Nikolaos Bellas, Ibrahim N. Hajj, Constantine D. Polychronopoulos, George Stamoulis, "Architectural and Compiler Techniques for Energy Reduction in High-Performance Microprocessors", in IEEE transaction on Very Large Scale Integration (VLSI) Systems, Vol. 8, No.3, pp 317-326, June 2000.

[10]Johnson Kin, Munish Gupta, William H. Mangione-Smith, "Filtering Memory References to Increase Energy Efficiency," in *IEEE Transactions on Computers*, vol. 49, no. 1, pp. 1-15, Jan., 2000.

[11]Weiyu Tang, Rajesh Gupta, Alexandru Nicolau, "Design of a Predictive Filter Cache for Energy Savings in High Performance Processor Architectures," in IEEE International Conference on Computer Design (ICCD'01) pp.68, 2001.

[12]Ramesh Panwar, David Rennels, "Reducing the frequency of tag compares for low power I-cache design" in International Symposium on Low Power Electronics and Design, pp. 57-62, 1995.

[13]Parthasarathy Ranganathan, Sarita Adve,Norman Jouppi, "Reconfigurable Caches and their Application to Media Processing", in International Symposium on Computer Architecture (ISCA2000), pp 214-224, 2000.

[14]M. Powell, Se-H. Yang, B. Falsafi, K. Roy and T. N. Vijaykumar, "Gated-Vdd: a circuit technique to reduce leakage in deep-submicron cache memories", in International Symposium on Low Power Electronics and Design (ISLPED), pp. 90–95, 2000

[15]S.-H. Yang, M. D. Powell, B. Falsafi, K. Roy, and T. N. Vijaykumar. Dynamically resizable instruction cache: An energy-efficient and high-performance deep-submicron instruction cache. Technical Report ECE-007, School of Electrical and Computer Engineering, Purdue University, 2000.

[16]D. Tarjan, S. Thoziyoor and N. Jouppi, "CACTI 4.0: An Integrated Cache Timing, Power and Area Model", Technical report, HP Laboratories Palo Alto, June 2006.

[17]T. Austin, E. Larson and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling", IEEE Computer, vol. 35, n. 2, pp. 59–67, February 2002

[18]S.Wolf, Silicon Processing for the VLSI era Volume 3 – The submicron MOSFET. Latice press, 1995, pp. 213-223.

[19]C. Zhang, F. Vahid and W. Najjar, "A highly configurable cache architecture for embedded systems", in International Symposium on Computer Architecture (ISCA), pp. 136–146, 2003.

[20]J. Kin, Munish Gupta, W.H. Mangione-Smith, "The filter cache: an energy efficient memory structure" in 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97), pp. 184, 1997.

[21]Kugan Vivekanandarajah, Thambipillai Srikanthan, Saurav Bhattacharyya, "Dynamic Filter Cache for Low Power Instruction Memory Hierarchy," in Euromicro Symposium on Digital System Design (DSD'04), pp. 607-610, 2004.

[22]A. Agarwal, J. Hennessy and M. Horowitz, "Cache performance of operating system and multiprogramming workloads", ACM Transactions on Computer Systems, vol. 6, n. 4, pp. 393–431, 1988.

[23]C. Zhang, F. Vahid and R. Lysecky, “A self-tuning cache architecture for embedded systems”, ACM Transactions on Embedded Computing Systems, vol. 3, n. 2, pp. 407–425, 2004.

[24]Chunho Lee, M. Potkonjak, W.H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *micro*, p. 330, 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97), 1997

[25]Arnold, B. *Embedded System Design*. CMP Books, 2002

[26]DineroIV Trace-driven Uniprocessor Cache Simulator, Mark D. Hill, Jan Edler, http://pages.cs.wisc.edu/~markhill/DineroIV/

[27]Hai Li, Amit Agarwal, Yiran Chen, Kaushik Roy, “DRG-Cache : A Single Vt Low Leakage Cache for Deep Submicron”. SRAM report.

[28]Moinuddin K. Qureshi, Yale N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *micro*, pp. 423-432, 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), 2006

[29]Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., Joel Emer, "Set-Dueling-Controlled Adaptive Insertion for High-Performance Caching," *IEEE Micro*, vol. 28, no. 1, pp. 91-98, Jan/Feb, 2008

[30]Afrin Naz, Krishna Kavi, Juan Oh, Pierfrancesco Foglia, “Reconfigurable Split Data Caches: A Novel Scheme for Embedded Systems”, Proceedings of the 2007 ACM symposium on Applied Computing.