

ABSTRACT

Title of Thesis: ELIMINATING INTER-PROCESS CACHE INTERFERENCE THROUGH CACHE RECONFIGURABILITY FOR REAL-TIME AND LOW-POWER EMBEDDED MULTI-TASKING SYSTEMS

Degree Candidate: Rakesh Reddy
Degree and Year: Master of Science, 2007

Thesis directed by: Professor Peter Petrov
Department of Electrical and Computer Engineering

This study proposes a technique which leverages data cache reconfigurability to address the problem of cache interference in multitasking embedded systems. Modern embedded systems often implement complex applications, comprising of multiple execution tasks with heavy memory requirements. Data caches are necessary to provide the required memory bandwidth. However, caches introduce two important problems for embedded systems. Cache outcomes in multi-tasking environments are difficult to predict, thus resulting in very poor real-time performance guarantees. Additionally, caches contribute to a significant amount of power. We study the effect that multiple concurrent tasks have on the cache and, subsequently, propose a technique which leverages work on reconfigurable cache architectures to eliminate cache interference and reduce power consumption using application specific information. By mapping parallel tasks to different cache partitions, inter-task interference is completely

eliminated with minimal performance impact. Furthermore, both leakage and dynamic power is significantly improved.

ELIMINATING INTER-PROCESS CACHE INTERFERENCE
THROUGH CACHE RECONFIGURABILITY FOR
REAL-TIME AND LOW-POWER EMBEDDED
MULTI-TASKING SYSTEMS

by

Rakesh Reddy

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
Masters of Science
2007

Advisory Committee:
Professor Peter Petrov, Chair/Advisor
Professor Bruce L Jacob
Professor Manoj Franklin

© Copyright by
Rakesh Reddy
2007

Acknowledgments

First of all I would like to thank my advisor, Dr. Peter Petrov for all his time, help and knowledge. He was always willing to help and provided invaluable guidance in my research. I would also like to thank Dr. Jacob and Dr. Franklin for agreeing to be on my committee.

I am grateful for the research work I was able to pursue during my undergraduate studies and would like to extend my appreciation to Dr. Edward Lee, Dr. Srinivas Tadigadapa and Dr. Raj Rajkumar for giving me the opportunity to do research work with them and their advice. I would like to show my gratitude to Xiangrong Zhou for his help with my research. Also I would like to thank my professors and teachers over the years.

I would like to thank my brothers at Sigma Phi Epsilon Penn Theta for all the good times and Beth McNicol for all her help as I would not be in Maryland without her. I would especially like to thank Lindsay Holliday for keeping me sane for all these years.

Most importantly, I would like to thank my loving parents, Raghu and Amrutha. Thank you for always being there no matter what the situation was. I am where I am only through everything you have done for me and would be nowhere without you. I will never be able to do anything to truly show my appreciation and gratitude but know that I will never forget what you have done.

Table of Contents

List of Tables	iv
List of Figures	v
List of Abbreviations	vi
1 Introduction	1
1.1 Embedded System Characteristics	2
1.2 Multitasking in Embedded Systems	4
1.3 Methodology	8
2 Related Work	10
2.1 Interference	10
2.2 WCET in the presence of caches	12
2.3 Reconfigurable Caches	15
3 Inter-Task Cache Interference	17
3.1 Observable Cache Interference	17
3.2 Evaluating the Observable Cache Interference	18
4 Cache Partitioning for Interference Elimination	28
4.1 Configurable Caches	29
4.2 The Cache Partitioning Problem	36
4.3 Relaxed Partitioning	40
4.4 Design Flow for Cache Partitioning	41
5 Cache Partitioning Evaluation	43
5.1 Performance Impact	43
5.2 Impact on Dynamic and Leakage Power	47
6 Conclusion	50
6.0.1 Future Work	50
Bibliography	52

List of Tables

3.1	Multi-task benchmark sets	20
3.2	Overall miss rate (percentages)	20
5.1	Overall miss rate (percentages)	43
5.2	Partition for Benchmark 7, 32kB 8 Way cache	44

List of Figures

3.1	Observable cache interference. a) Single task b) A task pre-empted then resumed	18
3.2	Global and local caches for interference miss classification . . .	18
3.3	Trace Format	22
3.4	Interference misses for 16KB configurations	25
3.5	Interference misses for 32KB configurations	26
4.1	Cache misses for a) ADPCM Decode b) MPEG2 Encode c) Lame Encode d) MMUL	30
4.2	Way Shutoff [1] pg 251	31
4.3	Way Concatenation [2] pg 374	32
4.4	Set Selection hardware	33
4.5	Drowsey cache line circuitry [3] pg 2	34
4.6	Cache partitioning examples	36
4.7	Heuristic partitioning algorithm	39
4.8	Exploration of partition space	39
4.9	Design flow of the cache partitioning methodology	41
5.1	Difference of miss-rate for strict partitioning vs. base configuration	45
5.2	Difference of strict partitioning and overlapped partitioning vs. base configuration	46
5.3	Dynamic power reduction	47
5.4	Leakage power reduction	48

List of Abbreviations

CPI Cycles per Instructions DSP Digital Signals Processor PID Process IDentifier MMU

Chapter 1

Introduction

Embedded systems are omnipresent in today's world. More and more products look to embedded systems to allow for increased functionality that was prohibitive when using traditional technologies such as logic and pure analog circuits. General purpose processors that are found in desktop systems are too costly and overkill in addressing the design issues of computing in everyday devices. Embedded processors have stepped in to fill this gap to allow for cost-effective implementation of computing in modern products. These processors are found in a plethora of areas including consumer electronics, consumer products, industrial processes, automobiles and wireless sensor networks.

Modern embedded systems have become increasingly complex as they find their way into increasingly demanding applications. Embedded applications impose demanding requirements for performance as they need to handle various data processing functions. For example current cell phones must be able to handle speech coding for communication, encoding pictures taken via on-board camera, decoding pictures to display images, decoding and encoding video streams, compressed audio decoding for music playback and user interface control. To meet these demands, modern embedded processors have evolved, and in the process borrowed many concepts from high-performance

general-purpose microprocessors.

The memory hierarchy is one of these concepts, addressing the problem of the growing discrepancy between memory and processor speeds. Caches are used to approach this increasing gap. Caches are placed between the processor and main memory. Caches are smaller in size than main memory but have much faster access times to reduce latency. The key to caches is the exploitation of locality. Locality can be classified as temporal or spacial. Temporal locality is the case in which data that has been recently accessed will be accessed again in the near future. Spatial locality is the tendency for data in a certain part of the address space is needed in clusters. Caches store the accessed data and data around it when main memory is accessed. By doing this, the latency to access data can be reduced if the desired data is in the cache. This speed up comes at the cost of increased power, in some cases as much as 50% of the total chip power [4] and reduced predictability.

1.1 Embedded System Characteristics

While embedded processors have borrowed many concepts from general purpose systems there is a conflict in design goals. In general purpose processors, the main design goal is better performance in terms of average execution time. To approach this problem more hardware resources can be added. For example, pipelines are increased to overlap more instructions or branch prediction is used to determine which branch will be taken to achieve faster execution.

While performance is important, meeting power, cost, and size constraints are just as important, if not more so in embedded systems. Furthermore, it is the instantaneous performance which is more important. The instantaneous performance is the amount of execution time everytime a task is run. Often times this time varies so the worst case execution time is used as a upper bound. Hardware such as increased pipeline stages and branch prediction may improve performance but also results in increased power consumption and chip cost. Characteristics of embedded systems include:

Low power Low power is often essential for two reasons in embedded systems. Embedded systems are often employed in mobile devices powered by battery and low power is key in allowing devices such as PDA's and cell phones to operate for long periods of time. In some cases such as wireless sensors, devices are deployed with limited access and must be able to operate as long as possible on a single charge. Furthermore, high power consumption also lead to increased heat which can cause to processor to become unreliable. This problem can be addressed with cooling solutions but this would add to the size and cost of the system.

Low Cost Products using embedded systems are often in markets where cost is a major concern. Also these devices can be produced in the millions so even small increases in cost can lead to large losses in revenue. Designers must work with very constrained resources to keep products costs down.

Predictability/Reliability Because of the real-time burden that often exists in these systems, the execution time must predictable. It is necessary

that the system is well behaved and predictable under all events and conditions. Embedded systems are often placed in critical systems such as industrial control and automotive applications. For example, anti-lock braking systems monitor slippage in car traction. If a deadline to compute the slip amount is not met, the entire system could fail and lead to a disastrous outcome. For this reason average performance has little importance as it is not predictable. If there is a great deal of variance within average performance, a real-time system must conservatively assume that it executes with the worst time possible.

1.2 Multitasking in Embedded Systems

Moreover, market demands require combined functionality of many application domains including multimedia processing (speech, image, and video), wireless capabilities, security features and user interfaces as mentioned above. The nature of many of these applications also requires that they are processed in real-time as a part of their specification. For example on-line speech processing algorithms must meet deadlines. If deadlines are not met in this case, end-users will notice delays in speech and reduced quality. A dedicated processor such as a DSP or multiple discrete chips could be used for each task. However, such a solution is often impractical as it results in increased power consumption, layout size and cost. Instead it is advantageous to execute multiple tasks on a single processor as it results in superior hardware utilization. Recent embedded processors have offered hardware support for multitasking,

such as Memory Management Units (MMU). Embedded OSes have also become readily available to utilize this hardware and support multitasking. The need for real-time performance has also led to the wide availability of real-time operating systems to ensure execution schedules where deadlines are met. A multitasking system must address several issues that are not relevant for a single task system. One such issue is that during task preemption, the preempted task must preserve its state so it may properly resume execution regardless of the activities of the preempting task. This involves saving information such as the PC, stack pointer and register file. For these smaller hardware structures, this is fairly simple and can be accomplished in a reasonable amount of time. Saving and reloading the state of the cache to memory for every task, however, is infeasible due to the large cache size. As an alternative, the cache is shared between the tasks without preserving its state.

Virtual memory is an elegant solution in making complex tasks such as memory allocation, memory sharing and protection transparent to applications with support from the OS and memory management hardware [5]. With the increase in multi-tasking virtual memory has found its way into embedded systems. Tasks access virtual addresses which are mapped to physical memory. To accelerate the translation from virtual to physical memory, many processors employ the Translation Lookaside Buffer which acts as a cache for recently translated addresses. The use of virtual memory leads to several design issues in caches. The operations in a cache access can be classified as indexing and tagging. Each of these operations can be implemented using either physical or

virtual memory [6]. In a virtually-indexed, virtually tagged cache, accesses can be made without translating the address. This reduces the latency and power consumption incurred in having to look up the physical translation. This form of caching suffers from the problems of aliasing and synonyms. Aliasing is when the same virtual address of multiple tasks map to different physical addresses. When accessing the cache a task may falsely hit on another tasks tag in the cache. There are two solutions to this problem. The first is to flush the cache on every context switch but this has a significant impact on performance. Alternatively, the tag can additionally store Process ID (PID) which is unique to each task. This information can be used to ensure that a task hit on it entry in the cache and not that of another task. The problem of synonyms occurs when to tasks share a physical address but map to it with different virtual addresses. This can lead to the situation in which multiple copies of the physical address are in the cache causing cache coherence problems if the data is modified. Solutions to this problem are complex and consume significant amounts of power restricting their usage in embedded systems. The opposite cache implementation is physically tagged and physically indexed caches. The problems of aliasing and synonyms are non-existent since all the physical addresses are unique and multiple copies can not exist. The drawback is that virtual to physical translation must be done on every access resulting in increased power and latency. Physically tagged and virtually indexed caches is a compromise between the above two methods. The tag translation is done in parallel with the index access to hide the latency. The OS and additional

hardware can be implemented to eliminate the synonym problem with no performance drawback.

Sharing the cache, however, leads to inter-task cache interference which is detrimental not only to performance, but even more importantly to real-time responsiveness. Cache interference occurs when a task block in the cache is overwritten by another task. General purpose processors can address this problem with increased cache sizes to reduce the likelihood of data in the cache being evicted. On the other hand, embedded systems are resource constrained thereby precluding an increase in the cache size. Cache interference can be very problematic for several reasons. Interference complicates *Worst-Case Execution Time (WCET)* analysis. The purpose of the WCET is to identify an upper bound on the tasks execution time to ensure predictability. Unlike general purpose systems, many real-time applications must meet deadlines based on WCET in order to operate properly. This analysis is complex, but well researched in the case of a single task [7, 8, 9], but predictability with multiple tasks using the cache becomes extremely complex, if not impossible, and usually results in a overly pessimistic analysis and under utilization of the processor. Additionally, interference increases the miss-rate of a task running alone versus running within a group of other tasks. With more tasks there is an increased likelihood that a task's data is overwritten and more misses occur. Both problems can be alleviated with more hardware, but this results in increased energy and cost and as such is infeasible for embedded

1.3 Methodology

While embedded systems have become more complex, the set of applications that they run is well defined during development compared to their general purpose counter part. To address the inter-task cache interference problem, we have performed a detailed analysis on the effect of cache interference in a multitasking system. Moreover, we introduce a methodology which leverages configurable caches where the data cache is judiciously partitioned so that each task has its own part of the cache which other tasks do not affect. We determine this partitioning by analyzing the cache behavior of a given set of applications. Identifying the best partitioning of the cache amongst the task is performed during compile-time and the information regarding the cache partitions, which consists of control signals to the configurable cache is transferred to the OS when loading the tasks. During context-switch the OS configures the cache by activating the cache partition of the preempting task. The proposed technique has two key benefits. First, techniques used for WCET analysis for a single task using a cache can be used since inter-task interference is eliminated. This allows for much better WCET analysis, and therefore better processor utilization. Second, by using reconfigurable cache architectures, significant reductions in dynamic and leakage power is achieved as only a portion of the cache is active at any time. All these benefits are achieved with minimal impact on the total miss rate as compared to the baseline where all the tasks share the cache. For some groups of tasks the total

miss rate is minimally increased, while for others it is decreased due to the interference elimination.

Chapter 2

Related Work

2.1 Interference

The effect of cache interference due to multitasking has been shown to have significant effect on performance. Agrawal et al [10] use a trace driven simulation in order to study the affect multitask workloads on cache performance for general purpose system. Cache interference was studied based on miss rates of tasks running alone and together for various cache configurations. Both the effect of the system code on user code and multiple user tasks are studied. In regard to system and user code interaction, it is shown that system code causes significant degradation in performance. Splitting the cache between user and system space performed worse than a shared cache. This study should be taken with a grain of salt as the systems simulated are fairly old and have a large system footprint. It was also shown the effect cache flushing on context switch performs to avoid inadvertent use of data is worse than using PIDs in virtual caches.

In [11] the authors have focused on the effect interference has on context switches in a general purpose multi-tasking system. Again traces are generated and fed into a cache simulator. This study sampled the CPI for some number of instructions following a context switch. This is then compared to

the CPI of the task running alone over this interval. It is shown the cost of interference in the cache is comparable, if not worse, than many other costs that are associated with context switching such as saving and restoring the register file. Furthermore, it is shown that increasing the cache size reduces this penalty and reducing the frequency of context switches can also improve performance.

The recent growth of multiprocessor systems has lead to many studies of multiple tasks affect on shared caches. Interference in multithreaded and multichip systems has become a very important topic and several have proposed various solutions [12, 13, 14]. Chandra et al developed heuristic and analytical approaches to study cache contention between tasks in the L2 cache. The *stack distance* is profile for each task running alone. The stack distance profile captures temporal reuse of an application. Prediction and probability models were created to use this information in order to estimate the additional misses that occur due to inter-task interference. Their analytical model based on *inductive propability* estimates the additional interference with an average error of 3.9%. It is likely that this model would be very inaccurate for the L1 cache because of the significant amount of increased accesses and dynamic nature at the L1 cache compared to the L2 cache.

Wang et al approach the problem with a thread-associative cache in which each ways is further grouped into *rails*. Each rails corresponds to a specific thread. In essence all this does is split the cache equally amongst threads.

In regards to embedded systems, the effect of interference posed on responsiveness in a real-time embedded system was recently analyzed [15]. This study looks at how various RTOS tasks services such as clock tick handler, context switch routine, mutexes and message passing are affected by various tasks due to cache interference. Certain tasks can cause significant miss rates for these services leading to unpredictability and a less responsive system. However, no analysis was presented on how multiple tasks would interfere with each other.

These studies show that interference in caches can not be neglected. However, all of these studies look at interference based on the affect on system performance but not interference explicitly. None of these studies provide an accurate categorization of misses due to interference or misses that would inherently occur with no other tasks running.

2.2 WCET in the presence of caches

Architectural features used to improve performance such as pipelining, branch prediction and caches make WCET much more difficult. WCET is a critical component in embedded systems as there are often real-time constraints that must be met. Unlike general purpose systems, operating in a timely manner is a necessity for correctness. Having a more accurate knowledge of the WCET allows for better utilization of the processor. Heckmann et al [7] employ abstract interpretation to analyze the WCET of an application

on architectures that implement pipelining, out-of-order execution, caches and branch prediction. This static analysis approach executes on the control flow of the graph to determine cache accesses. *Must Analysis* and *May Analysis* are used to predict if an access can be guaranteed to be a hit or a miss over a part of the control flow of an application. These two approaches are used to create upper and lower bounds on accesses results. Furthermore, the impact of pipelining and its effect on the cache is considered. In a conditional branch, each path can have a different affect on the cache. The damage to the cache is determined to be the worst case between the two flows of execution. It also shown that analyzing WCET with the interdependencies between different processor componenets is much more precise.

While WCET analysis with single tasks is difficult, the unpredictability of caches with multiple tasks is even more formidable. Interference in the cache between tasks leads to the pessimistic assumption that a task's data is invalid after a context switch. Several approaches propose solutions that place restrictions on preemption [16, 17] which may be undesirable for many applications. One method to achieve more accurate WCET analysis is to partition the cache so tasks are restricted to a subset of the cache. This eliminates the conflicts between tasks thereby ensuring better predictability. There have been hardware and software approaches to this method. Software based approaches employ the compiler to map tasks to only certain parts of the cache [18, 19]. To achieve this, code must be transformed so a task maps to only certain portions of the cache. For the instruction cache, the end of

each partition must have an unconditional jump to the next partition since each memory mapping must be non-linear to ensure a task does not use another task's partition. Also changes in control flow across memory mappings requires additional instructions. In context of the data cache, the compiler must add more code for structures that exceed the partition size. For example, an array that fits does not fit in a memory mapping can not simply be indexed since the compiler must map it in a non-linear fashion in memory to ensure partitions are maintained. These approaches fail to realize any savings in power and neither study looks at the impact on performance and code size due to the transformations.

In [20] the data cache is equally partitioned and each set of tasks with the same priority level are mapped to a shared equally-sized portion of the cache. Tasks of the same priority share cache, hence only interference from higher priority tasks is considered. It is shown that there is an impact on performance because of extra misses from restricting an application's cache space but performance is significantly better than eliminating the cache. Cache locking is used to try to minimize the degradation in performance but no detailed methodology is proposed on how to partition the cache is given. In [21] a priority based scheme for a unified cache is proposed which focuses on worst case response time for higher level tasks. Cache lines deemed important for a task are locked in the cache. While these studies improve WCET, they do not use information on tasks cache behavior which can cause significant increase the miss-rate.

2.3 Reconfigurable Caches

The goal of reconfigurable caches has been to reduce the power consumption of the cache by configuring it based on the behavior of a task. Certain tasks can perform just as well with only a subset of the cache resulting in an unnecessary consumption of power from the underutilized portion. Depending on the technique used, power savings can be achieved on dynamic or leakage power. To address this problem, several contributions have been recently made in the area of reconfigurable cache architectures [1, 22, 23, 2]. Disabling associativity ways has been shown to be very efficient in significantly reducing dynamic power [1]. In [2] a scheme is proposed that uses disabled ways combined with concatenating ways and varying block sizes. In [23] the cache is configured by varying the sets that can be accessed to reduce leakage in the unused portion. The hardware details of these implementations will be discussed in greater detail in section.

While several architectures are proposed, there has been few little work on how to configure the cache. One approach has been to use hardware to dynamically tune the cache for application based on its miss rate [24, 25]. In both papers, this approach entails counters that track the number of misses over a certain interval. The cache is than reconfigured and misses are tracked again over the specified interval. This new miss rate is compared to the previous to determine the configuration. Different heuristics are used to determine how to retune the cache including decreasing the cache size while the miss rate

remains below a specified threshold or an estimation of how the total energy for on an off chip memory dissipates. The number of cache configurations can grow very quickly as more configuration options are presented which n lead to extended periods of time where the configuration is in a suboptimal state.

Hu et al employ a software based technique to reduce data interference within an application[26]. *Compiler directed cache polymorphism* is used to determine an optimal configuration for loop nests to reduce energy with out impacting performance. The compiler analyzes data reuse within the nested loops of a single application and the footprint of arrays for nests is determined. This information used to determine cache configurations that can be tuned to focus on reducing power or improving performance. The method is restricted to statically declared arrays in nested loops and requires some code restructuring.

The methods to leverage reconfigurable caches so far are limited to looking at a single task running alone. A software based way partitioning scheme is proposed in [27] based on assigning statically higher priority multiple tasks more ways. No evaluation of the approach is made. The approach we propose aims at configuring the cache amidst multiple tasks running at the same time with the objective of eliminating interference and as such make multitasking with cache sharing a feasible approach for real-time and energy-efficient embedded applications.

Chapter 3

Inter-Task Cache Interference

3.1 Observable Cache Interference

Inter-process cache interference occurs when a cache line belonging to one task is replaced by another task, which prevents the first task from finding its data in the cache. We define cache interference to be only those misses that occur as a result of another task evicting a block which would not have occurred if the task was running alone and as such would have found the data in the cache. A task causes an interference miss only if it evicts a block that will be used by another task once it resumes as opposed to the task resuming and missing for other reasons. Figure 3.1a shows the memory accesses of a single task and Figure 3.1b shows a task being preempted. In the single task scenario, Task 1 reads A resulting a in cold miss, followed by hits on the following reads of A. In Figure 3.1b, Task 1 is preempted by Task 2. Task 2 reads B causing A to be evicted and D causing C to be evicted. When Task 1 resumes execution and reads A, it results in a miss because of Task 2's execution. Since this would not have occurred had Task 1 not been preempted, we classify this as an observable interference miss. Note that a preempting task evicting the preempted tasks block is not sufficient for for a interference miss. The preempted task must use the line that was evicted to be considered

Normal cache simulations can not determine if an access is an interference miss because this conclusion relies on future information of whether the block will be used again and not evicted by the owner task.

In our evaluation of interference misses we have developed a simulation-based approach, which precisely identifies whether a cache miss is due to an interference or not. Our approach in determining if a miss is the result of interference is to assign each task its own local cache which only it has access to, and a global cache, as shown in Figure 3.2. The global cache acts as a cache normally would with all tasks accessing it and potentially interfering with each other. Each task also has its own local cache that only it accesses. The local cache in essence stores the cache line's liveness state because the only way it can be evicted from the local cache is if the task evicts it itself. For every access, a task accesses the global cache and its local cache with each returning a hit or a miss. Based on the results of these caches, the access can be categorized as follows:

Global Hit, Local Hit. An access that hits in both the global and local cache. This corresponds to a normal cache hit.

Global Miss, Local Miss. An access that misses in both the global and local caches. This signifies an access that is a miss regardless of whether or not there were other tasks and hence does not contribute to interference and is treated as a normal miss.

Global Miss, Local Hit. An access in which the local cache access hits while the global cache access misses. Since the block is still in the local cache

	Task 1	Task 2	Task 3	Task 4
Bench 1	LAME Encode	ADPCM Decode	-	-
Bench 2	Matrix Mult	JPEG Decode	-	-
Bench 3	MPEG2 Decode	GSM Encode	-	-
Bench 4	ADPCM Decode	JPEG Encode	EPIC Encode	-
Bench 5	ADPCM Encode	GSM Decode	G721 Decode	-
Bench 6	MPEG2 Encode	GSM Encode	G721 Encode	-
Bench 7	ADPCM Decode	GSM Decode	Matrix Mult	JPEG Decode
Bench 8	MPEG2 Decode	G721Encode	GSM Encode	GSM Decode
Bench 9	EPIC Encode	JPEG Encode	G721 Decode	ADPCM Encode
Bench 10	Lame Encode	EPIC Encode	GSM Decode	ADPCM Decode

Table 3.1: Multi-task benchmark sets

Cache	16KB 4 Ways		16KB 8 Ways		32KB 4 Ways		32KB 8 Ways	
Switch Time	33k	100k	33k	100k	33k	100k	33k	100k
B1	1.87	1.73	1.86	1.73	1.31	1.29	1.23	1.24
B2	2.60	2.35	2.57	2.31	2.12	2.28	2.02	2.27
B3	0.11	0.11	0.07	0.07	0.07	0.07	0.05	0.05
B4	1.47	1.44	1.40	1.46	1.12	1.13	1.10	1.11
B5	0.04	0.04	0.02	0.02	0.02	0.02	0.02	0.02
B6	0.12	0.12	0.09	0.10	0.10	0.10	0.08	0.08
B7	2.40	1.89	2.45	1.86	1.86	1.83	1.66	1.83
B8	0.16	0.16	0.09	0.10	0.10	0.09	0.05	0.05
B9	1.10	1.04	1.06	1.06	0.79	0.80	0.76	0.77

Table 3.2: Overall miss rate (percentages)

and is being read it is still alive but was prematurely evicted in the global cache due to interference. We will refer to this miss type as an *interference miss* for the rest of this paper.

Global Hit, Local Miss. A global cache hit and a local cache miss situation is impossible if there is no data sharing because a task’s data in the global cache is always a subset of that in the local cache. If tasks are sharing data, this situation corresponds to one of the tasks “prefetching” the data for the other.

The memory traces were created using the SimpleScalar [29] simula-

tion infrastructure. SimpleScalar is a cycle-accurate processor simulator. The toolset includes varying degrees of simulation from basic instruction execution with not timing analysis to an out-of-order processor with timing, branch prediction and cache information. SimpleScalar supports several instruction sets including Alpha, ARM, x86 and PISA. This study uses the PISA instruction set using the *sim-cache* tool which simulates a cache. Unfortunately, SimpleScalar does not support any form of multi-tasking. To simulate multi-tasking, SimpleScalar was modified to generate memory traces from applications. Information generated includes the address of the memory access, whether it was a read or a write, and the time between memory accesses. The time between access is maintained to track the execution progress in terms of the total instruction executed. Figure 3.3 shows the format of the output. There are two types of trace lines which are generated. The first is the simple case in which the most significant bit classifies the access as a read or a write, the next 7 bits correspond to the number of instructions since the last memory access and the remaining 4 bytes is the address. In some cases the time between memory accesses was more than what could be represented in 7 bits. To address this, an extended trace line format was created. For this line the most significant byte is a 0. This value is guranteed to not conflict with the simple trace line format since this byte contains the number of instructions since the last access which must be non-zero (counting the second access itself). The next bit determines if the access was a read or a write, followed by 15 bits for the last access and the remaining four bytes the address. While 15 bits for the

Standard Trace Line

Field	R/W	Last Access	Addr
Bits	39	38:32	31:0

Extended Trace Line: Last Access Overflow

Field	Extnd: (0)	Op: (0xFF)	R/W	Last Access	Addr
Bits	71:64	63:56	55:48	47:32	31:0

Figure 3.3: Trace Format

last access count could have been used for trace lines, the need for this many bits was very rare and would have significantly increased the size of the traces which were already fairly large.

The memory traces created by SimpleScalar were then fed into a custom cache simulator which was verified against the Dinero cache simulator[30]. The custom simulator supports varying cache size, block size, and associativity ways. Entries were additionally tagged with the process ID for each task to avoid any conflicts between tasks. To simulate multi-tasking, the simulator reads a task's trace until a certain number of instructions were executed. The instructions executed is determined from the time since last access information of each trace entry. Once the specified number of instructions are executed, the task is preempted and the simulator starts reading the next task's trace in the same manner. This is continued with each task being scheduled in a round robin policy until all tasks have completed.

The cache simulator creates a global cache and local cache for task as describe above. For every access, the global cache and the task's local cache are accessed. The access is then classified as a normal hit, normal miss or interference miss based on the results of the cache accesses.

Memory traces for applications from the MediaBench [31] and the MiBench [32] benchmarks suits were generated. Below is a description of the benchmarks used. These applications are fairly representative of applications that are found in current cell phones and mobile media devices.

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: An image data compression utility based on a biorthogonal critically-sampled dyadic wavelet decomposition and a combined run-length/Huffman entropy coder.

ADPCM: ADPCM stands for Adaptive Differential Pulse Code Modulation. It is a family of speech compression and decompression algorithms. A common implementation takes 16-bit linear PCM samples and converts them to 4-bit samples, yielding a compression rate of 4:1.

GSM: A full-rate speech transcoding algorithm using residual pulse excitation/long term prediction. The implementation turns frames of 160 16-bit linear samples into 33-byte frames.

G721: Reference implementation of the CCITT (International Telegraph and Telephone Consultative Committee) G.721 speech coding specification

MMUL: A basic 128x128 element matrix multiplication.

The traces are subsequently used to simulate individual tasks and multi-tasked benchmarks with cache sizes of 16KB and 32KB sizes and associativities

of 4 and 8 ways which reflects current embedded processors such as the *Intel XScale* and the *ARM9*. The block size is fixed at 32 bytes. For multi-tasking scheduling, a simple round robin approach with fixed execution slices of 33,000 instructions and 100,000 instructions are studied to look at how the frequency of context switches affect interference. These values correspond to 1ms for 33MHz and 100MHz with a CPI of 1.

All the tasks were executed until completion and their memory and execution progress traces captured. By grouping together tasks we have constructed various multitasking benchmarks, comprising of 2, 3, and 4 parallel tasks. In this study we evaluate the cache interference between the application tasks and do not include any kernel code. The context-switch kernel code has a very small data footprint and, if need be, can be assigned to its own very small partition of the cache (or bypass the data cache altogether). Complex kernel operations are not common for embedded applications and, if required, the kernel can be treated as yet another task in the group, which uses the data cache and possibly introduces interference. The kernel cache behavior is very specific to the OS and its particular implementation; it could differ significantly even across different versions of the same OS. In this paper we focus on the task interference. Table 3.1 shows the combinations of tasks used for each multitasking benchmark and Table 3.2 shows the overall miss rates.

Figures 3.4 and 3.5 report the misses for each benchmark with the above mentioned configurations and the interference encountered by each task. The crossed out parts of the bars correspond to interference misses. There are

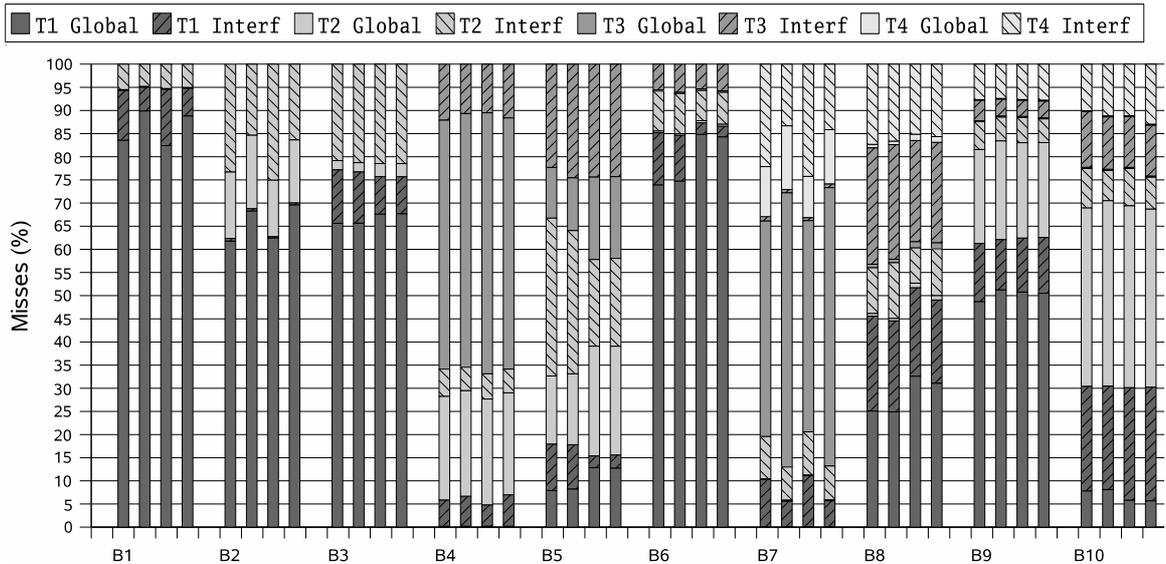


Figure 3.4: Interference misses for 16KB configurations

several distinct behaviors among the applications. ADPCM, GSM and G721 suffer from significant amounts of interference but have relatively low miss rates. These applications exhibit strong temporal and spatial locality and as a result there is an increased propensity that data evicted due to preemption will be used again. As a result these applications suffer a great deal from interference. On the other hand JPEG, EPIC and LAME, have relatively high miss rates to begin with so the affect of interference is relatively small. This high miss rate also mitigates the impact of interference on other task as show in B1, B4 and B9 of figure 3.4. These applications have higher miss rates since they do not exhibit much locality. As a result, the liveliness of these blocks is low and evicting these tasks blocks is not as likely to cause interference. However, large amounts of data are brought in, thus increasing the interference seen by other tasks. These applications are also cache starved which will be shown later in this paper. Another behavior is exhibited by the streaming

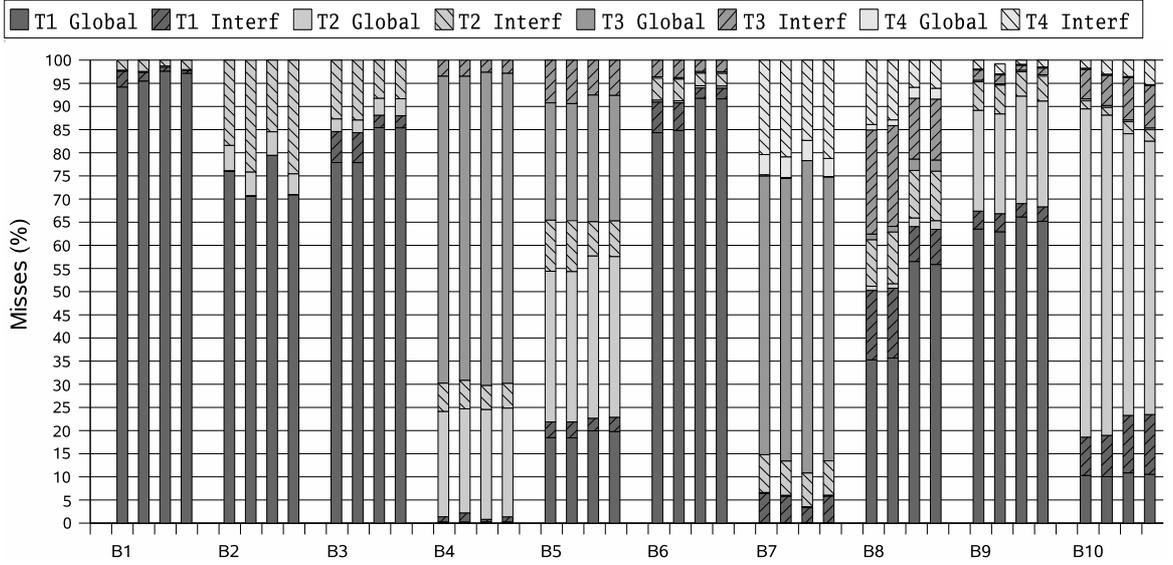


Figure 3.5: Interference misses for 32KB configurations

applications. A streaming application is one that shows limited temporal locality with very good spacial locality. These applications incur a large miss rate but are not impacted by other applications. Streaming applications differ from the likes of JPEG and EPIC in the fact that they are not cache starved. A side effect of the poor temporal locality of these applications is that they create increased amounts of interference in other applications as they bring large amount of data without reusing it. Matrix Multiply (MMUL) acts similar to a streaming application. While it is not normally considered a streaming application, it requires a cache much larger than those studied to successfully exploit temporal locality. For most of the multitasking benchmarks the lower context switch frequency results in higher interference since cache block liveness usually decreases with time. The exception to this is MMUL. As MMUL acts much like a streaming application, when this type of application is allowed to run for longer periods of time, it increases the likelihood that another task's

data is evicted thereby increasing interference.

In general, the larger 32kB cache configurations suffer less interference. The larger cache decreases the likelihood that a cache block is evicted since there is more space. More tasks usually resulted in a larger percentage of interference because more tasks would be competing for space. This, however, depends is highly dependent on the set of tasks because some tasks will cause more interference than others. A higher switching frequency also increase the amount of interference since tasks bring more data in that will interfere more often. One interesting observation is that higher set-associativity does not always reduce the percentage of interference. This can be attributed to the decrease in total misses resulting in interference misses becoming a larger percentage.

Chapter 4

Cache Partitioning for Interference Elimination

The above figures have shown that multiple tasks sharing the cache can exhibit a significant amount of cache interference. For some benchmarks, interference misses account for over 50% of the total misses resulting in significant degradation in performance compared to tasks running by themselves. What is worse is the loss of predictability in the system. Even for the best case benchmark, 10% of the misses are attributed to interference. This effect can not be ignored and must be considered in WCET. In the case where real-time requirements must be met, ignoring interference will likely yield a system that does not meet its specification. Alternatively, conservative approaches would mean the system is not being fully utilized. We address this problem by partitioning the cache so each task is limited to a non-overlapping portion of the cache with a size tuned to the task needs. We refer to this partitioning scheme a strict partitioning scheme since we allow no overlap. To minimize this, we partition the tasks based on their cache behavior. By ensuring that tasks share no parts of the cache, it is guaranteed that no interference occurs. This makes the system much more predictable and easier to analyze for WCET and when task cache behavior is used, we can do so with little or no impact on performance and in some cases even improve performance. Furthermore, we

evaluate an extension of the strict partitioning, which we refer to as overlapped partitioning, where some tasks share their cache partitions.

4.1 Configurable Caches

A configurable cache is advantageous over conventional caches because it can be fine-tuned to a specific task. With multiple tasks running on a single processor, certain tasks may require a smaller cache size than others for acceptable performance. This forces the designer to use a chip with a larger cache. To some tasks the extra cache provides minimal or no benefit at the cost of increased power consumption. With configurable caches, this cost can be reduced by disabling portions of the cache for tasks that show marginal benefit from having access to the full cache.

Figure 4.1 shows how various cache configurations affect applications. The figures depict several applications and their misses with varying ways and set sizes which correspond to cache configurations that they can be mapped to. In Figure 4.1a, we see that the miss rate for ADPCM decode saturates fairly quickly and increasing the cache size after a certain point has no effect. MPEG2 decode in Figure 4.1b shows a similar behavior. Increasing the cache ways or set size starts to converge and after a certain point only minimal improvements are achieved even when the cache is doubled. We refer to this as the point of minimal gain. The LAME codec in Figure 4.1c has a continually downward slope and none of the set lines converge signifying it could still

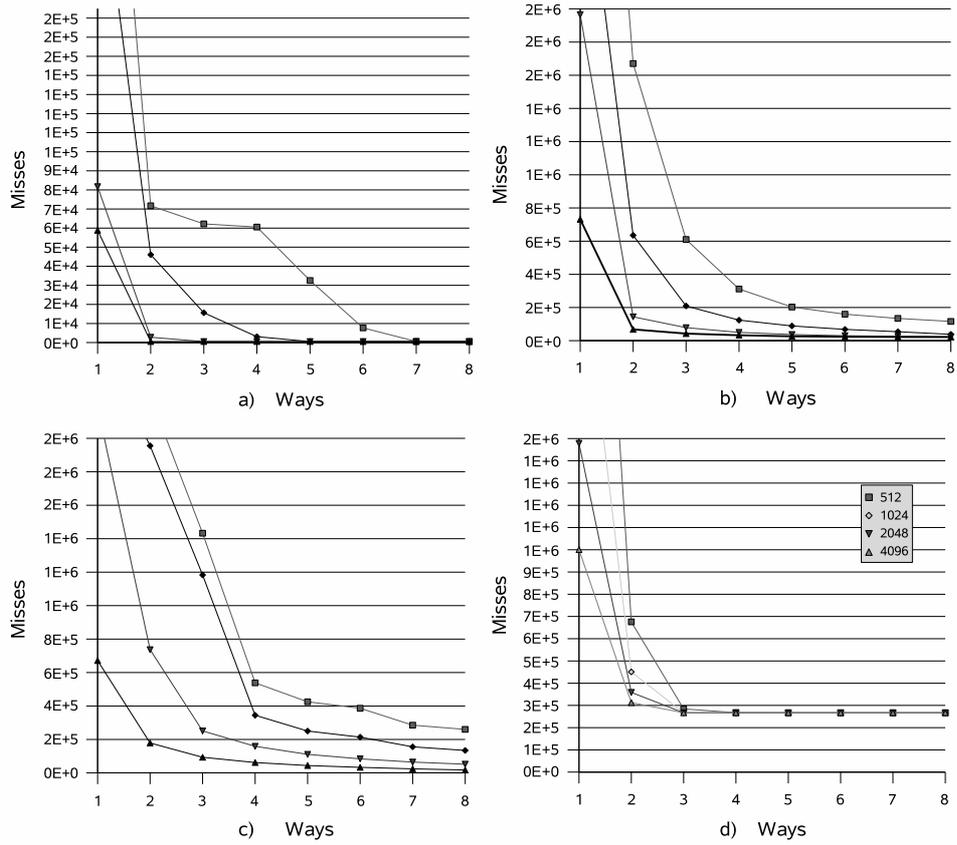


Figure 4.1: Cache misses for a) ADPCM Decode b) MPEG2 Encode c) Lame Encode d) MMUL

benefit from increased cache. MMUL in Figure 4.1d shows an application that shows a significant amount of misses with increases in cache size having almost no benefit. This is because the cache size is too small to exploit significant amounts of temporal locality.

Several techniques for cache reconfiguration exist in order to reduce the power consumption of caches. Chip power consumption can be classified into two categories, dynamic and static. Dynamic power corresponds to activity on the processor, predominantly from the charging or discharging the capacitive load when switching a gate. Static power on the otherhand is power consumed

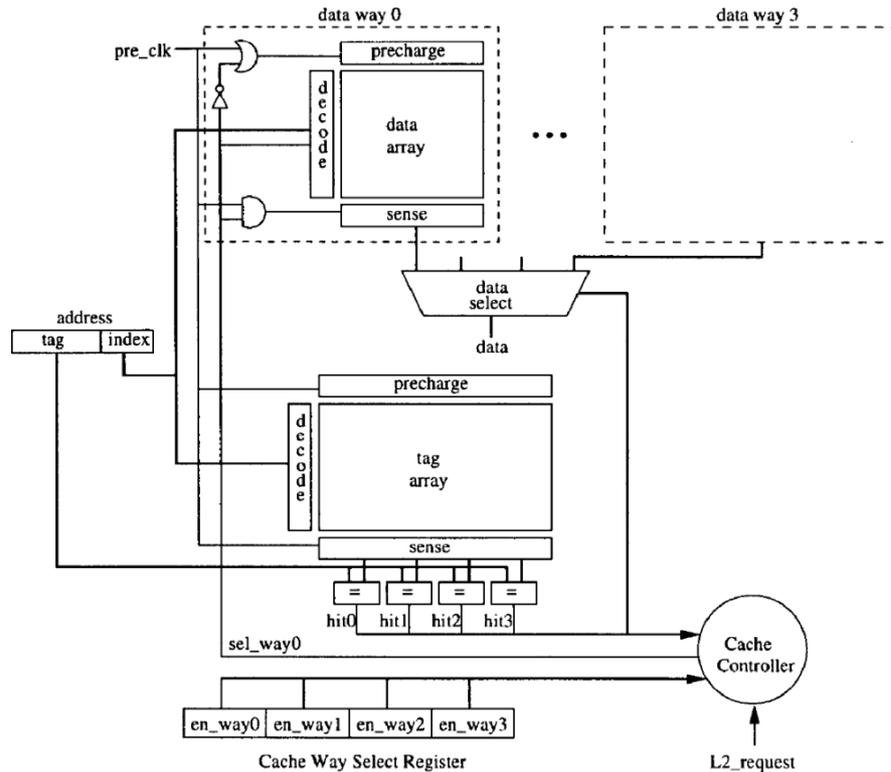


Figure 4.2: Way Shutoff [1] pg 251

simply when the chip is on. The most significant form of static power is subthreshold leakage current which is the current that leaks between the source and drain terminals of a transistor and increases with shrinking feature size [33].

The architecture proposed in [1] selectively disable ways to reduce the dynamic power. Registers are configured by software to control which ways are enabled as shown in figure 4.2. The *cache way select register* is used to determine which ways to shutdown. The pre-charge signal is gated so when the cache is accessed, ways that are disabled are not pre-charged and data in these ways are not read. This reduces pre-charge of bitlines which is one of the main factor the causes dynamic power consumption in caches. The selected

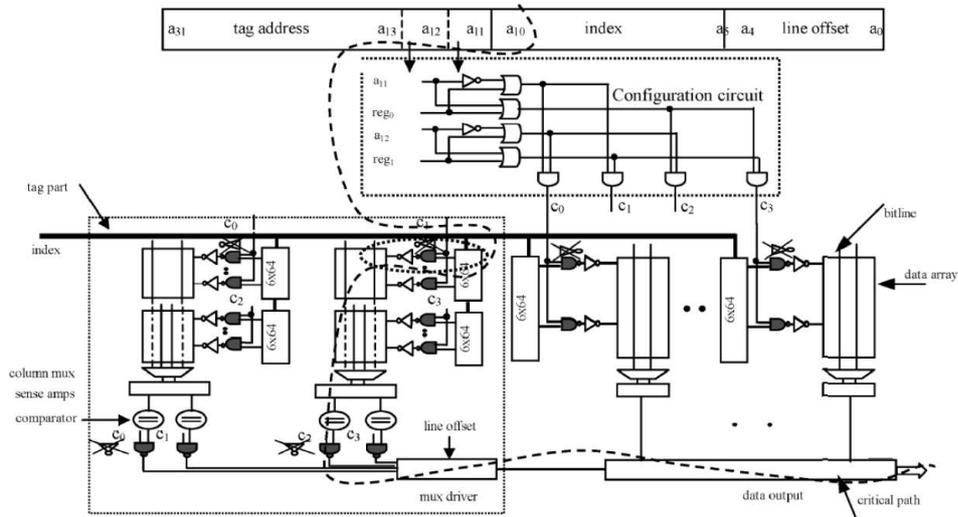


Figure 4.3: Way Concatenation [2] pg 374

ways information is also sent to the cache controller to ensure that a hit in the tag lookup of a disabled way is ignored.

The configurable cache proposed in [2] introduces a hardware that allows for configurations of the associativity, caches size and block size. Their work presents the idea of way concatenation in which ways are combined to make larger sets. Ways are concatenated by using a bit from the index to select which ways read and limiting dynamic power to these ways as shown in 4.3. When $reg1=1$ and $reg0=1$ the cache acts like a normal 4-way set associative cache. If $reg1=0$ and $reg0=0$ the cache becomes direct mapped with the signal c_i determining which way will be accessed. When either $reg1=1$ and $reg0=0$, or $reg1=1$ and $reg0=0$, the cache becomes 2-way set associative and only two ways are accessed. For example a 4-way set associative cache with 256 sets could be changed to be a 2-way, 512 set cache or 1-way 1024 set cache by using one of the index bits to select which group of ways to access. As a result, for each concatenation, half the ways are read on any access reducing the dynamic

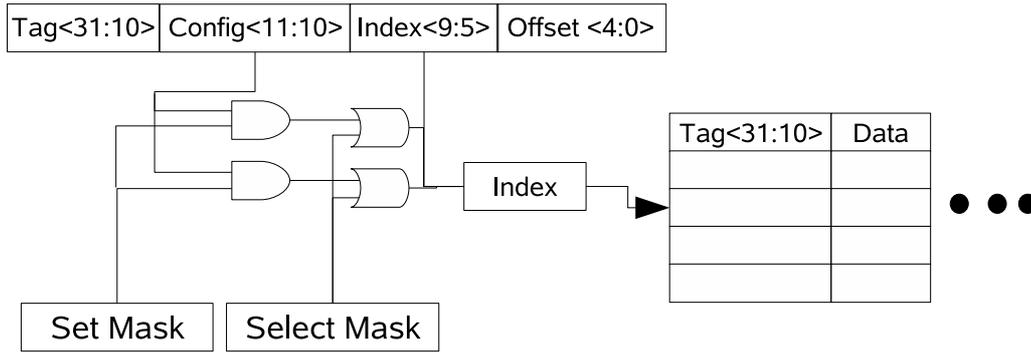


Figure 4.4: Set Selection hardware

power.

In [34] the number of sets that can be accessed by a cache is configurable and the use of gated- V_{dd} is proposed to reduce leakage power in disabled sets. Gated- V_{dd} introduces a transistor between the SRAM cell and ground reducing leakage based on the “stacking effect” of transistors in series. The number of sets can be configured in powers of two by masking the number of bits used in the index. Extra tag bits must be used since decreasing the number of sets requires a larger tag for correctness. For example, a cache with with 256 sets be changed to 128 and 64 sets with two extra tag bits. There are two potential deficiencies of this hardware for our approach. First, the hardware presented is limited in that it can only vary the set size but not which set can be used. However, this can be resolved through a simple manipulation of the cache index in a manner similar to [35]. Adding OR gates after the mask can select the group of lines that is being mapped to as shown in Figure 4.4. As described above the address is first masked to determine the size of the caches it is mapped to and then it is passed through a second set of gates to

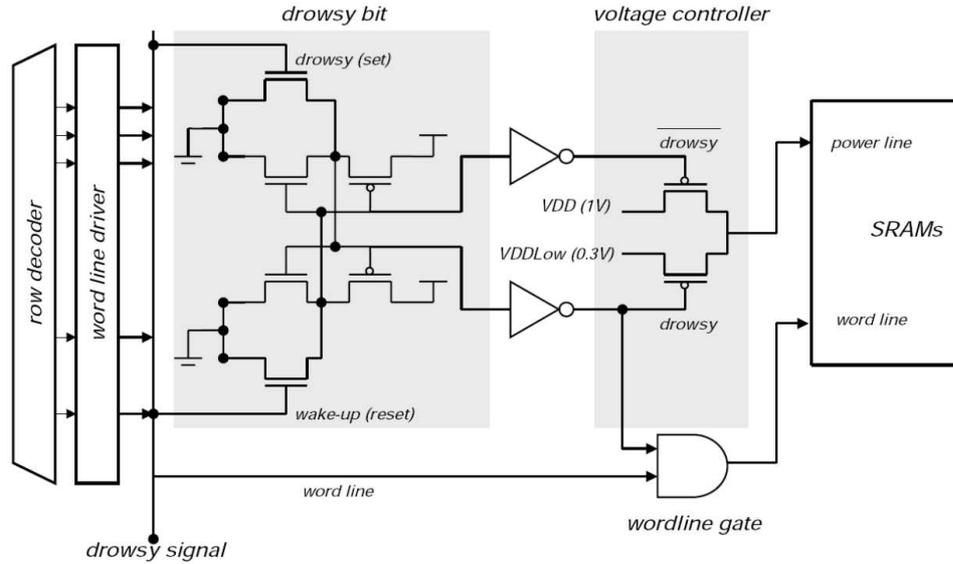


Figure 4.5: Drowsy cache line circuitry [3] pg 2

determine which portion of the cache it is mapped to.

While providing significant reduction in leakage current, gated V_{dd} does not maintain data in the cache resulting in cold start misses when it is turned back on. An alternative technique is to use drowsy caches [3]. Drowsy cache techniques place cache lines into a low power mode which reduces leakage but maintains data in the cache. Such a cache has been implemented in [3] by using dynamic voltage scaling which reduces leakage by a factor 12. By increasing the supply voltage, V_{dd} , to cache lines, leakage power is reduced. The additional circuitry for drowsy cache lines is shown in Figure 4.5. The main additions are a drowsy state bit, voltage controller and a wordline gate. The drowsy state bit determines whether a cache line is in active or drowsy mode and sets which voltage to use. The wordline gate ensures that data is not read when the cache is in a drowsy state because this data would likely be incorrect. If a cache line is not in a drowsy state, no performance penalty

is incurred. Lines in a drowsy mode will incur an increased latency penalty because the line state must be changed to active and reread. Discussion of drowsy caches in the rest of this paper refers to the implementation presented by Flautner et al.

Our study leverages the ideas in the above work. We vary the number of ways and sets used by each task and use the hardware to partition tasks thereby eliminating interference. The inactive parts of the cache can be placed in drowsy mode, thus reducing the cache leakage power. Drowsy caches do not reduce leakage as much as gated V_{dd} . However, we rely on the ability for data to be preserved in the disabled portions of the cache for more aggressive WCET analysis. We allow tasks to use any number of associativity ways as long as it is less than or equal to the base line configuration including non-power of two values. Shutting off ways requires a register containing a bit for each way and adds a gate to determine whether or not to pre-charge a way. In terms of set configuration, we assume that a cache partition can consist of either all the sets, half of them, or a quarter of them. Additionally, the selected set must be aligned at address boundary corresponding to their size. Set-selection requires a register that maintains the size of the partition and another which determines the portion of the cache to map to. This hardware lies on the critical path, however it is shown in [23] how this delay can be significantly minimized. Drowsy caches increases the cache line by 3% when implemented on a line by line basis. However, our approach requires only a coarse-level granularity for associativity-set partitioning. When drowsy cache

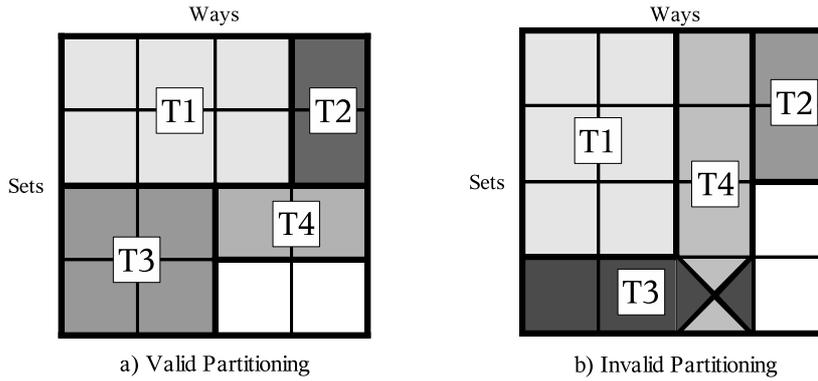


Figure 4.6: Cache partitioning examples

blocks are accessed in a non-drowsy mode, there is no impact on access time. Since in our approach drowsy lines are never accessed, the cache hit latency is not deteriorated. The time to turn drowsy cache lines back on is very small and can be done concurrently with other operations during the context switch.

4.2 The Cache Partitioning Problem

In the proposed partitioning scheme the cache is essentially divided into a set of rectangles, each consisting of a number of columns (ways) and a number of rows (contiguous group of sets). Each task configuration can similarly be abstracted to a rectangle and the set of all such rectangles must fit into the cache. Figure 4.6 depicts an example of such partitioning. The validity of a set of partitions is determined by the capabilities of the underlying configurable architecture. In this example we have a total of 4 ways and sets that can be configured down to a quarter of the original sets. Figure 4.6a shows a valid mapping of tasks. Note that none of the tasks overlap and that all set configurations are powers of 2 as previously discussed. The tasks are not

required to cover the area and in fact, covering less area while maintaining the number of misses equates to a even further reduction in power. Figure 4.6b shows an invalid partitioning that is due to several reasons. First T4 and T3 overlap making it invalid. Additionally, T1 is configured with a set size that is not a power of 2.

A static off-line approach is used to determine cache partitioning for the given set of tasks. A run-time approach would require extra hardware which is often undesirable in an embedded system due to size and power costs. Also run-time approaches introduce additional unpredictability into the system. While partitioning for a single task in hardware is feasible [23, 2], partitioning multiple tasks is complex and infeasible to perform at run time because of the immense number of combinations as the number of tasks and configurability of the cache increases. Our approach also simplifies hardware and does not require suboptimal configurations that exist during the tuning stages often found in hardware approaches.

Partitioning the tasks can be viewed as a set coverage problem. For each task T_i , we have a partition P_i where i identifies the task associated to that partition (from 1 to the total number of tasks N). Each P_i is an equivalent to a rectangle that represents valid configuration as defined by the cache architecture. This optimization problem can be formally defined as:

$$\text{maximize} \left(\sum_{i=1}^N \text{UTIL}(P_i) \right)$$

where $\text{UTIL}(P_i)$ is the hit rate (utilization) of task T_i assigned to its cache

partition P_i . This is clearly the goal of the proposed technique, as we want to maximize the cache utilization after partitioning it amongst the tasks in the system. The set of partitions P must satisfy:

$$\begin{aligned}
 P_i \cap P_j &= \emptyset, \text{ for } i \neq j \\
 \sum_{i=1}^N \text{COST}(P_i) &\leq \text{COST}(\text{Cache}) \\
 \text{VALID}(P_i) &= \text{TRUE for all } i
 \end{aligned}$$

The first condition ensures that the partitions are non-overlapping. The second constraint specifies that the sum of all cache partitions must not exceed the total cache; here $\text{COST}(P_i)$ refers to the size partition P_i . The third condition simply constraints the cache partitions to what is implementable by the underlying configurable cache. This is a combinatorial optimization problem with exponential complexity as it is a form of the NP-complete *minimal set-cover* problem. Systems of 2 or 3 tasks combined with a limited number of cache configurations (in the tens) are feasible to solve through an exhaustive search. However, the complexity of the problem quickly rises with more configurations and tasks. To solve this problem we offer a heuristic algorithm. The pseudocode of this algorithm is outlined in Figure 4.7.

Our heuristic approach starts by setting all tasks to have the smallest partition possible and adding them to an active list. The solution space is explored as shown in figure 4.8 by the *GROW* function. We start from the smallest partition and first increase size, then associativity. Circles with the same color signify partitions that are in equal size while the white circles

```

1   $P = P_i$ , where  $P_i$  is partition of task  $T_i$ 
2  for all  $P_i = \text{Smallest Valid Partition}$ 
3  Set tolerance value  $T$ 
4  Add all  $P_i$  to ActiveList
5
6  while( ActiveList != Empty AND  $\text{COST}(P) < \text{COST}(\text{Cache})$ )
7    for( Active Pi)
8      if(  $\text{UTIL}(P_i) > \text{BASE}(P_i) * T$ )
9        Remove  $P_i$  from ActiveList
10     else
11       GROW( $P_i$ )
12     if( $\text{COST}(P) > \text{COST}(\text{Cache})$ )
13       // Partitions can no longer grow
14       Option 1: BREAK
15       Option 2: Decrease  $T$ ; Re-Iterate
16     if( ActiveList == Empty )
17       //Done or Improve Solution
18       Option 1: BREAK
19       Option 2: Increase  $T$ ; Re-Iterate
20  END

```

Figure 4.7: Heuristic partitioning algorithm

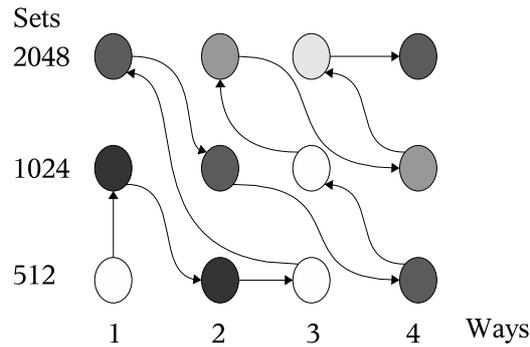


Figure 4.8: Exploration of partition space

have no equal partitions. Each partition is simultaneously grown until the tasks utility is greater than $\text{BASE} * T$. BASE function is the sum of normal and interference misses from the multitasking profile - it corresponds to the baseline configuration where all the tasks share the cache. The tolerance value $T \in [0 : 1]$ represents how close the task must be to the baseline hit-rate; a value of 1 enforces that the baseline hit-rate must be met or improved. Once a partition reaches this point, it is removed from the active list and becomes

associated to its corresponding task. If all partitions are removed from the task list then we have a configuration that performs within the tolerance interval of miss-rate impact for all the tasks. At every iteration the GROW function must check for partition validity as simply using less space than the entire cache does not guarantee validity.

4.3 Relaxed Partitioning

In many systems it is possible that only a subset of the tasks must meet real-time deadlines while other tasks are non-critical. For example, speech codecs must guarantee deadlines are met to ensure quality but image compression may occur offline. In this case, interference between non-critical tasks can be tolerated to provide larger partitions for more critical tasks.

The approach of relaxed partitioning is similar to strict partitioning with the difference that a subset of non-critical tasks is treated as a single task and assigned to a single cache partition. In essence, the main distinction is that we relax the policy of non-overlapping partitions. Tasks are divided into critical and non-critical tasks. Critical tasks are treated as before given their own partition. In the relaxed partitioning, non-critical tasks are assigned to share the cache partition. In this way applications with no real-time constraints but large memory footprints such as the mp3-encoder LAME, the video compression MPEG, and image compression JPEG, are associated to one large cache partition. The relaxed partitioning method allows us to focus resources on

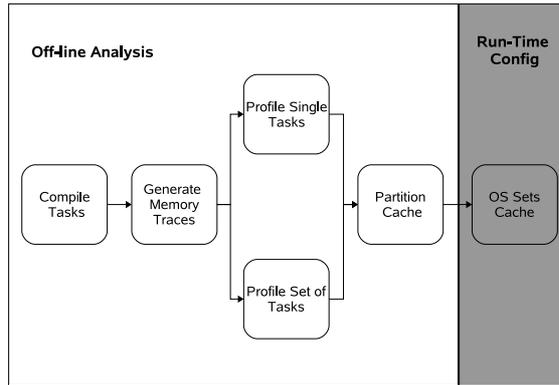


Figure 4.9: Design flow of the cache partitioning methodology

the most important tasks. To determine the partitioning for this scheme, we allocate partitions to tasks based whether they are critical or not. We again use a tolerance value in order to ensure that more than marginal gains are being achieved by increasing the cache partition. The set of non-critical tasks are treated as a single task and its cache utilization is profiled like a single task would be. In this way, the relaxed partitioning problem is reduced to the strict partitioning after which the heuristic presented above is used.

4.4 Design Flow for Cache Partitioning

Figure 4.9 shows the design flow of our approach. First, tasks are compiled and run through a memory trace generator. The use of traces allows for faster simulation and studying a larger design space. The traces contain information regarding both memory accesses and execution progress. Next, traces are profiled in two ways. The single task profiling consists of profiling the application with a cache simulator for all possible configurations allowed by the underlying cache architecture. The second profile is based on our approach

used to study interference - it provides the baseline miss-rate and interference statistics when the tasks share the cache. The last step performed off-line is the execution of the cache partitioning heuristic. This heuristic is executed on the set of task deemed to require separate partitions, possibly after merging the non-critical tasks to implement the relaxed partitioning scheme.

The final step comes in the run-time implementation of setting the control registers to configure the cache. The configurations for each task must be maintained by the OS to guarantee the cache is configured correctly during preemption. Since preemption occurs transparently to the tasks, if a task was preempted and the cache state is changed by another task, it would not know the cache needs to be reconfigured. Each task configuration could be stored in hardware but the overhead in performing the reconfiguration would be negligible. A w bits would be needed for w -ways and 2^s bits for set configuration (size mask and mapping). This amounts to a load and a reconfigure instruction that moves the information to the cache control registers. While we do not look at the interference of the OS, the kernel task could also be given a cache partition. As the embedded kernels do not exhibit large working sets a minimal cache partition dedicated to the kernel would often times suffice.

Chapter 5

Cache Partitioning Evaluation

We have evaluated the proposed cache partitioning techniques (strict and relaxed) on the set of multitasking benchmarks described in Section 3.2. We have profiled all the tasks for cache configurations covering all possible partitions including associativity sets of sizes 512, 1024, and 2048 and associativity ways from 1 up to 4/8 depending on the baseline cache architecture. Subsequently, the cache partitioning heuristic is performed with tolerance value $T=1$; if no valid solution for that value is found, the heuristic is re-executed with a relaxed value of T . Table 5.1

5.1 Performance Impact

Table 5.2 shows a cache partitioning for Bench 7. The first two columns are the total miss numbers for the shared cache. The partitioning for each task and the resulting misses are shown in the rest of the columns. In this case, all partitioned tasks perform better than when they share the complete cache.

Cache	Switch Time	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10
16KB 4-Ways	33,000	1.87	2.60	0.110	1.469	0.039	0.120	2.402	0.161	1.098	0.986
	100,000	1.73	2.35	0.110	1.441	0.038	0.119	1.889	0.162	1.044	0.949
16KB 8-Ways	33,000	1.86	2.57	0.074	1.404	0.024	0.094	2.453	0.090	1.055	0.970
	100,000	1.73	2.31	0.074	1.459	0.024	0.095	1.860	0.095	1.061	0.991
32KB 4-Ways	33,000	1.31	2.12	0.073	1.119	0.017	0.099	1.859	0.095	0.791	0.505
	100,000	1.29	2.28	0.073	1.130	0.017	0.097	1.832	0.093	0.799	0.518
32KB 8-Ways	33,000	1.23	2.02	0.054	1.100	0.015	0.084	1.658	0.048	0.762	0.587
	100,000	1.24	2.27	0.054	1.111	0.016	0.084	1.825	0.049	0.733	0.605

Table 5.1: Overall miss rate (percentages)

Tasks	Shared Cache Misses		Partitioned Cache		
	33k	100k	Ways	Set Size	Misses
Bench 7					
ADPCM D	13,176	25,277	3	2048	549
GSM D	29,634	33,311	3	1024	9,866
MMUL	266,917	267,244	3	1024	266,819
JPEG D	85,775	109,623	5	20480	43,907
Total	395,502	435,455			321,141

Table 5.2: Partition for Benchmark 7, 32kB 8 Way cache

This is because interference misses are eliminated. Also, as will be shown shortly, dynamic energy is reduced because less ways are accessed at anytime.

Figure 5.1 shows the absolute difference in miss rate of the strictly partitioned cache from the baseline cache for the various configurations. In most cases, the difference for the 8-way set associative caches is lower since it is more configurable than its 4-way counterpart. Partitioning on the 32KB cache is better than that for 16KB cache for every benchmark. Not only does the larger cache allow for more configurations, it also allows partitions to be closer to the point of marginal gains from increased cache. Benchmarks B1, B4, and B9 show increases in miss rate in all configurations. This can be attributed to the poor cache behavior of LAME, JPEG and EPIC encode. As discussed before these tasks are cache starved and partitioning forces the miss rate to increase significantly. This increase is especially high for caches with only 4-ways. Most of the opportunity for reconfiguration with a 4-way cache is by sets which is at a much larger granularity than adjusting ways. The high miss rate mitigates any gain from reducing interference. Our partitioning scheme performs very well for Benchmarks 2 and 7 because of MMUL streaming nature. By partitioning the cache, MMUL does not interfere with the other tasks,

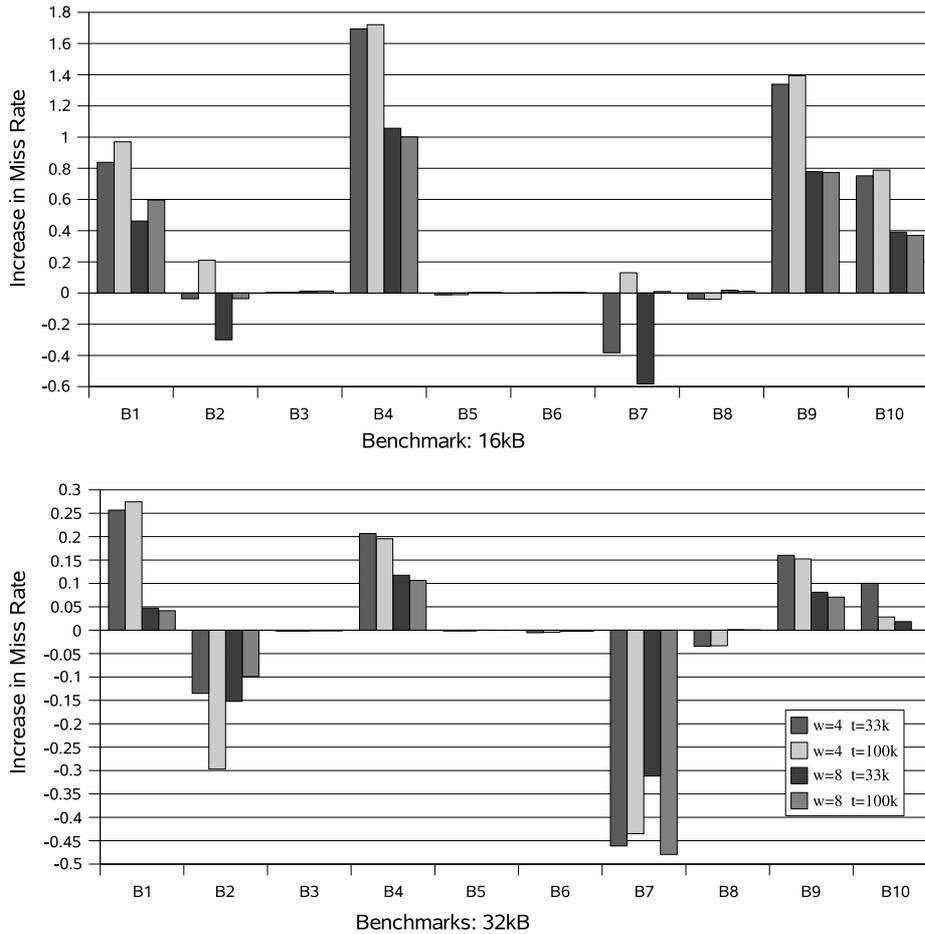


Figure 5.1: Difference of miss-rate for strict partitioning vs. base configuration where as if allowed to use the entire cache, it would significantly interfere with the contents of other tasks.

Figure 5.2 compares the difference between the strict partitioning and relaxed partitioning for each configuration for benchmarks B4, B9 and B10. The speech applications (ADPCM, GSM, G721) were classified as the critical tasks while JPEG, EPIC and MPEG2 were classified as non-critical. In general our overlapped partitioning technique had performance similar to the strict partitioning with the added benefit of increasing the response time of critical tasks. In many cases the overlapped partitioning also had better over-

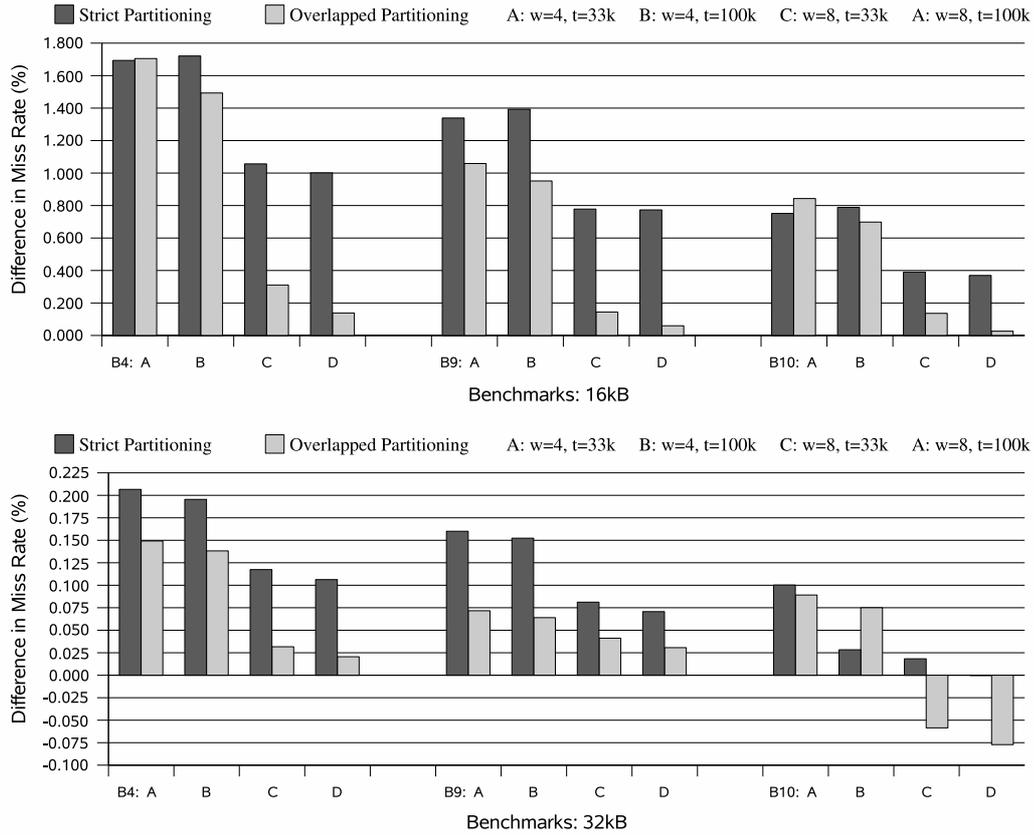


Figure 5.2: Difference of strict partitioning and overlapped partitioning vs. base configuration

all performance than simple strict partitioning. This can be attributed to the fact that the non-critical applications do not exhibit strong temporal locality. If applications that show poor temporal locality are grouped together, they can use a larger cache but interference will not have as large of an impact because of their cache behaviours. As a result the amount of interference in the non-critical applications is not as significant as the normal misses and the tasks benefit from the increased cache size offered by overlapping cache. This is consistent with our study of interference in which these applications suffered more from normal misses than interference. The amount of improvement over

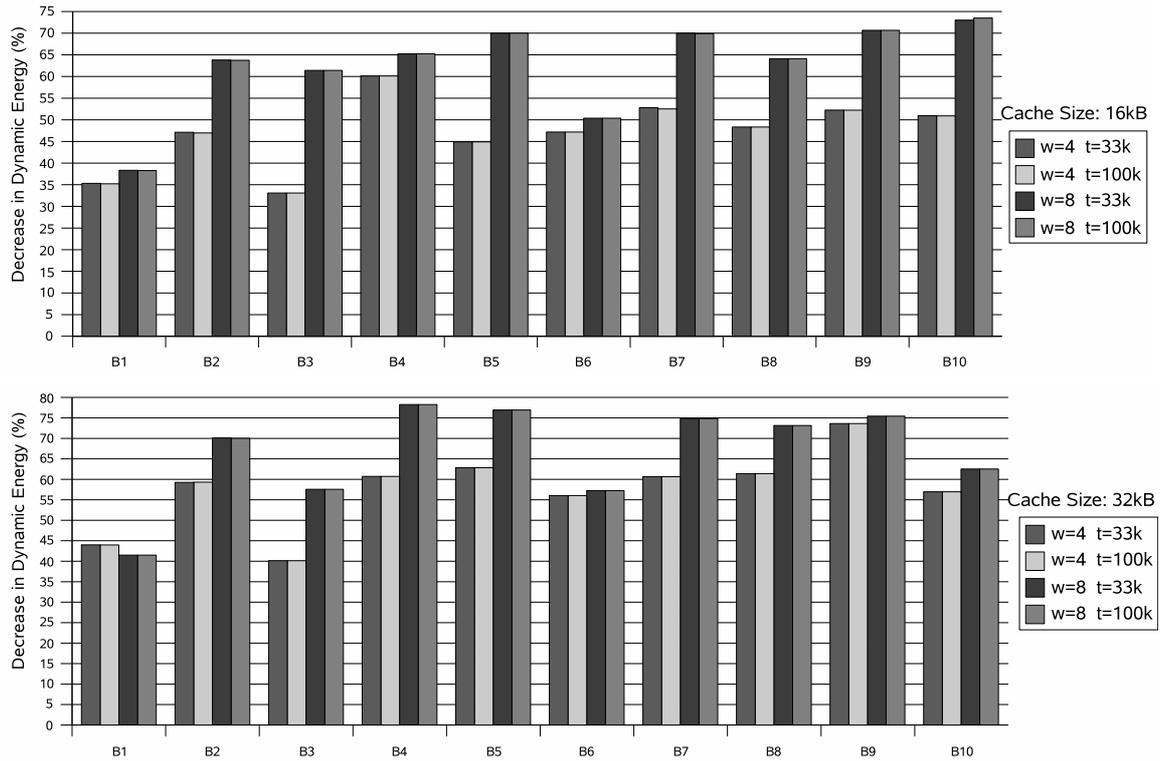


Figure 5.3: Dynamic power reduction

strict partitioning is not as significant in 4-way set associative caches because of the lack of configurability. The granularity of configurability can result in lower associativity caches using the same amount of cache one task would use between multiple tasks.

5.2 Impact on Dynamic and Leakage Power

We have evaluated the impact of the proposed cache partitioning technique on dynamic and leakage power. As only a single cache partition is active at any moment in time, both dynamic and leakage power are expected to be significantly reduced. The inactive parts of the cache are placed in drowsy mode in order to reduce the cache leakage power. Dynamic and leakage power were

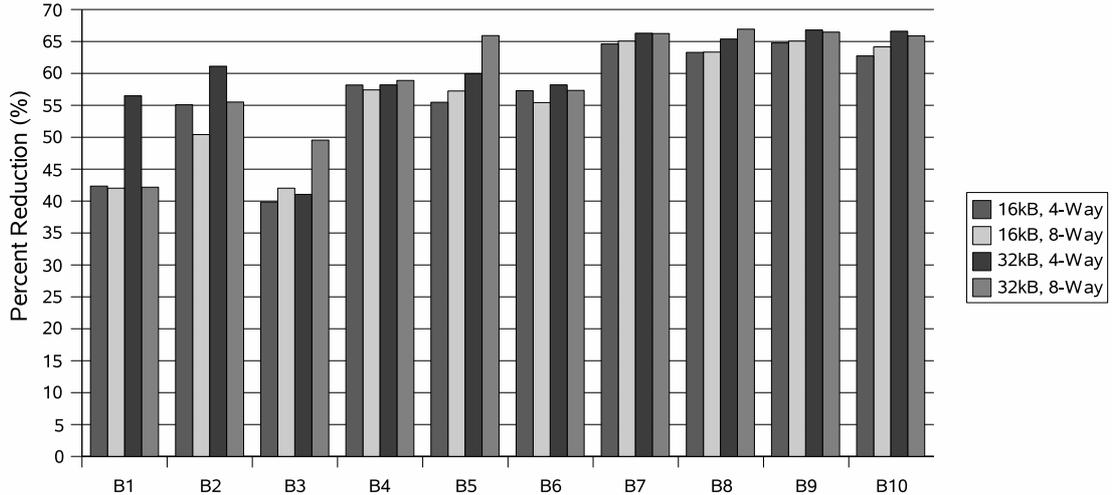


Figure 5.4: Leakage power reduction

modeled using Cacti-4.2 [36] with 180nm technology. Each cache partition was modeled as a separate cache and its power characteristics were obtained from Cacti. Because CACTI does not model associativities that are not a power of 2 this information was extrapolated. Caches misses were modeled as accesses to a 256KB direct mapped cache. Figure 5.3 shows the the data cache reduction in dynamic energy for the multitasking benchmarks after applying the proposed cache partitioning. The baseline configuration is all the tasks share the cache. Even for benchmarks where the miss-rate was slightly increased due to the partitioning we see a significant reduction in dynamic energy. In the worst case, dynamic power is still reduced by 30%. As one would expect 8-way set associative caches due to their higher power consumption show more improvement than 4-way associative caches.

The leakage power reduction is even more significant. In our evaluation of leakage power, we have assumed a drowsy cache implementation as proposed

in [3] controlled at granularity levels of associativity ways and the groups of associativity sets supported by our cache partitioning approach. Leakage power for the various cache partitions used in our multitasking benchmarks was obtained from Cacti-4. The inactive parts of the cache were assumed in drowsy mode and their leakage power reduced by a factor of 12 [3]. Figure 5.4 shows the leakage power reductions for our benchmarks. It can be seen that for all the benchmarks the leakage is reduced from 40% upto 65%. The benchmarks with 4 parallel tasks achieved consistently better leakage reductions, since with more tasks in the system, the cache had to be divided into smaller partitions which explains the trend of leakage reduction increasing with more tasks.

Chapter 6

Conclusion

In this paper we have introduced a novel methodology for inter-task cache interference elimination through data cache partitioning. Our methodology leverages recently proposed configurable cache architectures in order to assign the set of parallel tasks to non-overlapping cache partitions. We have outlined a compile-time algorithm, which uses profile-based information regarding the cache behavior of each task to identify a beneficial partitioning of the cache. The cache partition information for each task is provided to the OS, which during context-switch activates the cache partition of the preempting task while deactivating the one for the preempted task. Our results demonstrate that the proposed scheme not only eliminates data cache interference, thus making single-task WCET analysis algorithms applicable, but also significantly reduces both dynamic and leakage power of the data cache. The proposed cache partitioning enables the application of multi-tasking support with shared data caches in real-time and energy-efficient embedded systems.

6.0.1 Future Work

So far our study has assumed that tasks do not share memory. In the context of memory being shared between tasks, partitioning can cause cache

coherence problems. One possibility is to flush all dirty cache block on a context switch. This can cause significant degradation in performance and predictability. One way to minimize this penalty is to allow for a transition period after the context switch. During this period, a hardware mechanism can be used to write dirty cache blocks to memory. To ensure up to date data is used by the newly running task, lookups are made on the entire cache and forwarded accross partitions until all data has been written back to memory.

The partitioning scheme used can also be extended in many ways. Our current cost approach was to minimize the impact on performance. This can be extened to focus on reducing energy with a bound on performance degradation. Furthermore we restrict tasks to a single configuration during the course of their execution. Applications may show varying cache behavior accross different compute intensive portions of the task and this information can be used to reconfigure the cache within an application for reduced energy savings.

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] 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.
- [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] A. Malik, B. Moyer and D. Cermak, “A low-power unified cache architecture providing power and performance flexibility”, in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 241–243, 2000.
- [5] B. Jacob and T. Mudge, “Virtual memory: issues of implementation”, *IEEE Computer*, vol. 31, n. 6, pp. 33–43, June 1998.
- [6] M. Cekleov and M. Dubois, “Virtual-address caches. Part 1: problems and solutions in uniprocessors”, *IEEE Micro*, vol. 17, n. 5, pp. 64–71, September 1997.
- [7] R. Heckmann, M. Langenbach, S. Thesing and R. Wilhelm, “The influence of processor architecture on the design and the results of WCET tools”, *Proceedings of the IEEE*, vol. 91, n. 7, pp. 1038–1054, July 2003.
- [8] R. Kirner and P. Puschner, “Transformation of Path Information for WCET Analysis during Compilation”, in *Euromicro Conference on Real-Time Systems (ECRTS)*, page 29, 2001.
- [9] Y. S. Li, S. Malik and A. Wolfe., “Cache modeling for real-time software.”, in *IEEE Real-Time Systems Symposium*, pp. 148–157, 1997.
- [10] 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.
- [11] 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.
- [12] D. Chandra, F. Guo, S. Kim and Y. Solihin, “Predicting inter-thread cache contention on a chip multi-processor architecture”, in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

- [13] S. Wang and L. Wang, “Thread-associative memory for multicore and multithreaded computing”, in *International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 139–142, 2006.
- [14] G. Edward Suh, Srinivas Devadas and Larry Rudolph, “A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning”, in *HPCA*, pp. 117–, 2002.
- [15] J. Starner and L. Asplund, “Measuring the cache interference cost in preemptive real-time systems”, in *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 146–154, 2004.
- [16] M. Alt, C. Ferdinand, F. Martin and R. Wilhelm., “Cache behaviour prediction by abstract interpretation”, in *Static Analysis Symposium (SAS)*, pp. 52–66, 1996.
- [17] R. Arnold, F. Mueller, D. Whalley and M. Harmon, “Bounding worst-case instruction cache performance.”, in *Real-Time Systems Symposium (RTSS)*, page 172181, 1994.
- [18] A. Wolfe, “Software-based cache partitioning for real-time applications”, *Journal of Computer and Software Engineering*, vol. 2, n. 3, pp. 315–327, 1994.
- [19] F. Mueller, “Compiler support for software-based cache partitioning”, in *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 125–133, 1995.
- [20] X. Vera, B. Lisper and X. Jingling, “Data caches in multitasking hard real-time systems”, in *Real-Time Systems Symposium (RTSS)*, pp. 145–165, 2003.
- [21] Y. Tan and III V. J. Mooney, “WCRT analysis for a uniprocessor with a unified prioritized cache”, in *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 175–182, 2005.
- [22] H-C. Chen and J-S. Chiang, “A highly configurable cache architecture for embedded systems”, in *Communications, Computers and signal Processing (PACRIM)*, pp. 315–318, 2001.
- [23] S-H. Yang, B. Falsafi, M. D. Powell and T. N. Vijaykumar, “Exploiting Choice in Resizable Cache Design to Optimize Deep-Submicron Processor Energy-Delay”, *Symposium on High-Performance Computer Architecture (HPCA)*, vol. 00, pp. 0151, 2002.
- [24] 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.

- [25] M. Powell, Se-H. Yang, B. Falsafi, K. Roy and T. N. Vijaykumar, “An Integrated Circuit/Architecture Approach to Reducing Leakage in Deep-Submicron High-Performance I-Caches”, in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 147–157, 2001.
- [26] J. Hu, M. Kandemir, N. Vijaykrishnan and M. J. Irwin, “Analyzing data reuse for cache reconfiguration”, *ACM Transactions on Embedded Computing Systems*, vol. 4, n. 4, pp. 851–876, 2005.
- [27] K. Tanaka, “Fast Context Switching by Hierarchical Task Allocation and Reconfigurable Cache”, in *Innovative Architecture of Future Generation High-Performance Processors and Systems (IWIA)*, 2003.
- [28] Xiangrong Zhou and Peter Petrov, “Rapid and low-cost context-switch through embedded processor customization for real-time and control applications”, in *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pp. 352–357, New York, NY, USA, 2006, ACM Press.
- [29] 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.
- [30] M. D. Hill, “Test driving your next cache”, *MIPS Magazine*, pp. 84–91, August 1989.
- [31] C. Lee, M. Potkonjak and W. H. Mangione-Smith, “MediaBench: A Tool for evaluating and synthesizing multimedia and communications systems”, in *International Symposium on Microarchitecture (MICRO)*, pp. 330–335, December 1997.
- [32] M.R. Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, “MiBench: A free, commercially representative embedded benchmark suite”, in *WWC-4: Workshop on Workload Characterization*, pp. 3–14, December 2001.
- [33] Samuel Rodriguez and Bruce Jacob, “Energy/power breakdown of pipelined nanometer caches (90nm/65nm/45nm/32nm)”, in *ISLPED '06: Proceedings of the 2006 international symposium on Low power electronics and design*, pp. 25–30, New York, NY, USA, 2006, ACM Press.
- [34] 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.
- [35] P. Petrov and A. Orailoglu, “Towards effective embedded processors in codesigns: customizable partitioned caches”, in *9th International Symposium on Hardware/Software Codesign*, pp. 79–84, April 2001.

- [36] 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.