Eliminating Inter-Process Cache Interference through Cache Reconfigurability for Real-Time and Low-Power Embedded Multi-Tasking Systems

Rakesh Reddy
University of Maryland
College Park, USA
rnreddy@umd.edu

Peter Petrov
University of Maryland
College Park, USA
ppetrov@ece.umd.edu

ABSTRACT

We propose a technique which leverages configurable data caches to address the problem of cache interference in multitasking embedded systems. Data caches are often necessary to provide the required memory bandwidth. However, caches introduce two important problems for embedded systems. Cache outcomes in multitasking environments are notoriously difficult to predict, if not impossible, thus resulting in poor real-time guarantees. Additionally, caches contribute to a significant amount of power. These issues are key factors for many embedded systems. We study the effect of multiple tasks on the data cache, and propose a technique which leverages configurable cache architectures to eliminate inter-task cache interference. By mapping tasks to different cache partitions, interference is completely eliminated with only a minimal impact on performance. Furthermore, dynamic and leakage power are significantly reduced as only a subset of the cache is active at any moment. We introduce a profile-based, static analysis algorithm, which identifies a beneficial cache partitioning. The OS configures the data cache during context-switch by activating the corresponding partition. Our experiments on a large set of multitasking benchmarks demonstrate that our technique not only efficiently eliminates inter-task interference but also significantly reduces both dynamic and leakage power.

Categories and Subject Descriptors: B.3 [Hardware]: Memory structures; C.3 [Computer Systems Organization]: Special-Purpose and Application-Based Systems

General Terms: Algorithms, Design, Experimentation

1. INTRODUCTION

Modern embedded systems have become increasingly complex as they find their way into increasingly demanding applications. Embedded applications, such as cell phones and various hand-held devices impose strong requirements for performance as they need to handle various data processing functions such as speech, audio, and video. 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. This speed up comes at the cost of increased power, in some cases as much as 50% of the total chip power [15].

Moreover, market demands require combined functionality of many application domains including multimedia processing (speech, image, and video), wireless capabilities, security features and user interfaces. 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. A dedicated processor such as a DSP 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. 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.

Several solutions exist for allowing the tasks to share the data cache while maintaining correctness. 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. 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 [10, 12, 14]. However, in the case of multiple tasks sharing the cache, predicting hit/miss behavior is extremely complex, if not impossible, and usually results in a overly pessimistic analysis and under utilization of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CASES'07, September 30–October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-826-8/07/0009 ...$5.00.

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.

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. Furthermore, we introduce a methodology which leverages configurable caches where the data cache is judiciously partitioned so that each task has its own partition of the cache which is unaffected by other tasks. We determine this partitioning by analyzing the cache behavior for a given set of applications. Identifying the best partitioning of the cache amongst the tasks is performed during compile-time and the information regarding the cache partitions, consisting 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 with a cache can be utilized 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.

2. RELATED WORK

The effect of cache interference due to multitasking is a well known problem. In [1] cache interference was studied based on its affect on performance over an extended period of time for various cache configurations. Much of the focus of this paper was the interference between user and kernel space and evaluating the effect of cache flushing on context switch or using PIDs. In [16] the authors have focused on the effect interference has on context switches in a multi-tasking system by tracking the CPI. Both approaches take an aggregate count of interference based on performance as a whole and do not account for interference explicitly. Interference in multi-threaded and multichip systems has become a very important topic and various solutions have been proposed [6, 25]. In regards to embedded systems, the effect of interference posed on kernel services and the impact on responsiveness in real-time embedded systems was recently analyzed [20]. However, no analysis was present on how multiple tasks would interfere with each other.

The unpredictability of caches in WCET analysis with multiple tasks is an important problem. 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 rather than a convenience. Having a more accurate knowledge of the WCET allows for better utilization of the processor. 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 [3, 4] 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 [26, 17]. These approaches fail to realize any savings in power and neither study looks at the impact on performance. In [24] 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; furthermore, no detailed cache interference evaluation is presented. 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 significantly increase the miss-rate.

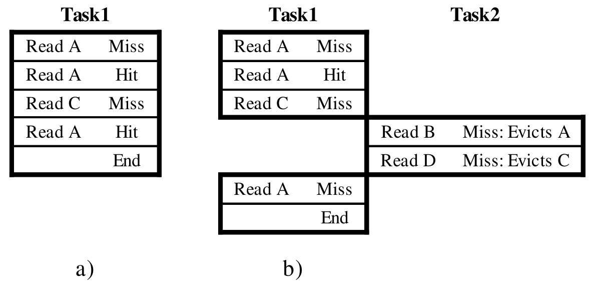

Figure 1: Observable cache interference. a) Single task b) A task preempted then resumed

The goal of configurable 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 [2, 7, 27, 29]. Disabling associativity ways has been shown to be very efficient in significantly reducing dynamic power [2]. In [29] a scheme is proposed that uses disabled ways combined with concatenating ways and varying block sizes. In [27] the cache is configured by varying the sets that can be accessed to reduce leakage in the unused portion. While several architectures are proposed, there has been few research projects on how to configure the cache. One approach is to use hardware to dynamically tune the cache for application based on its miss rate [28, 19] but this decreases predictability. A way partitioning scheme is proposed in [22] based on task priority but does not study the implementation. In [11] a software based technique is used to optimize loop nests within the application. The methods to leverage reconfigurable caches so far are limited to looking at a single task running alone. 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.

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

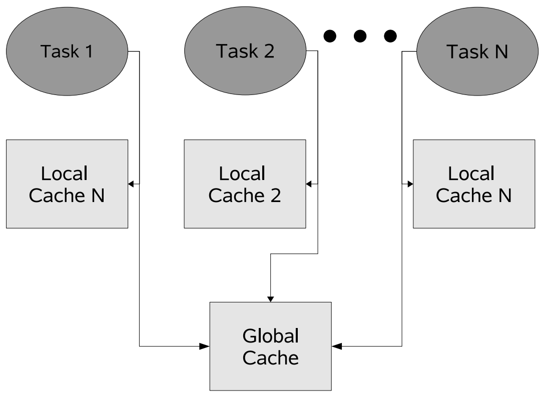

Figure 2: Interference miss classification infrastructure

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 1a shows the memory accesses of a single task and Figure 1b shows a task being preempted. In the single task scenario, Task 1 reads A resulting in a cold miss, followed by hits on the following reads of A. In Figure 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 an interference miss. In this case, Task 2 evicting C does not constitute an interference since Task 1 does not use this block again.

3.2 Evaluating Cache Interference

The difficulty in studying true interference in the cache can be attributed to the difficulty in knowing the life time of a cache line. That is to say, it is difficult to dynamically determine liveliness, or how long a cache line will remain in the cache and whether it will be used by the task later. The dynamic nature of data in the data cache with multiple tasks makes this impossible to study from an analytical perspective. 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 this is to assign each task its own local cache which only it has access to, and a global cache, as shown in Figure 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 liveliness 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 from these caches, the access can be categorized as:

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 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.

Using the SimpleScalar [5] simulation infrastructure we have generated memory traces for applications from the MediaBench [13] and the MiBench [9] benchmarks suits. The traces contain not only the memory references, but also information about the execution progress in terms of total instructions executed. The second piece of information is needed in order to model task preemption. 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 affected 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 1 shows the combinations of tasks used for each multitasking benchmark and Table 2 shows the overall miss rates.

Figures 3 and 4 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 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 shown in B1, B4 and B9 of figure 3. 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 applications. A streaming application is one that shows limited temporal local-

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

Table 1: Multi-task benchmark sets

Cache	Time Slice	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 2: Overall miss rate (percentages)

ity with very good spatial 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 liveliness 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.

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 or 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. 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 as strict partitioning scheme since we allow no overlap. 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 while in some cases even improve performance. Furthermore, we evaluate an extension of the strict partitioning, which we refer to as overlapped partitioning, where tasks with no real-time requirements are allowed to 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. 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 5 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 can be mapped too. In Figure 5a, 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 5b 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 5c has a continually downward slope and none of the set lines converge signifying it could still benefit from increased cache. Several techniques for cache reconfiguration exist.

The architecture proposed in [2] selectively disable ways to reduce the dynamic power. Registers are configured by software to control which ways are enabled. The configurable cache proposed in [29] 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. The way shut off is also used to allow for more configurability. In [18] 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. 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

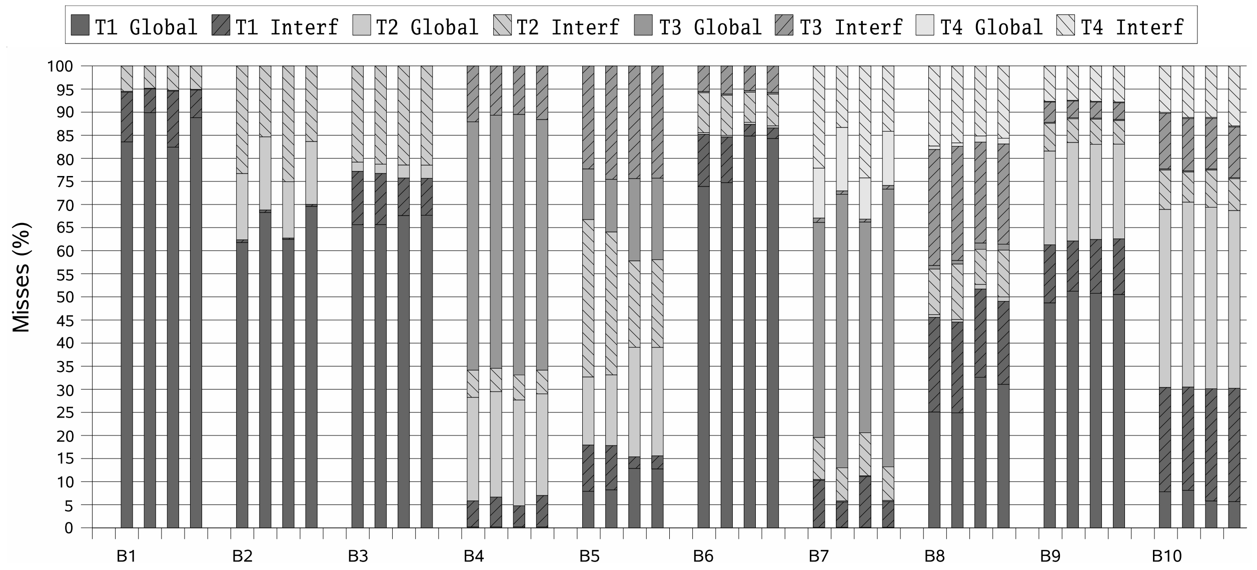

Figure 3: Interference misses for 16KB configurations

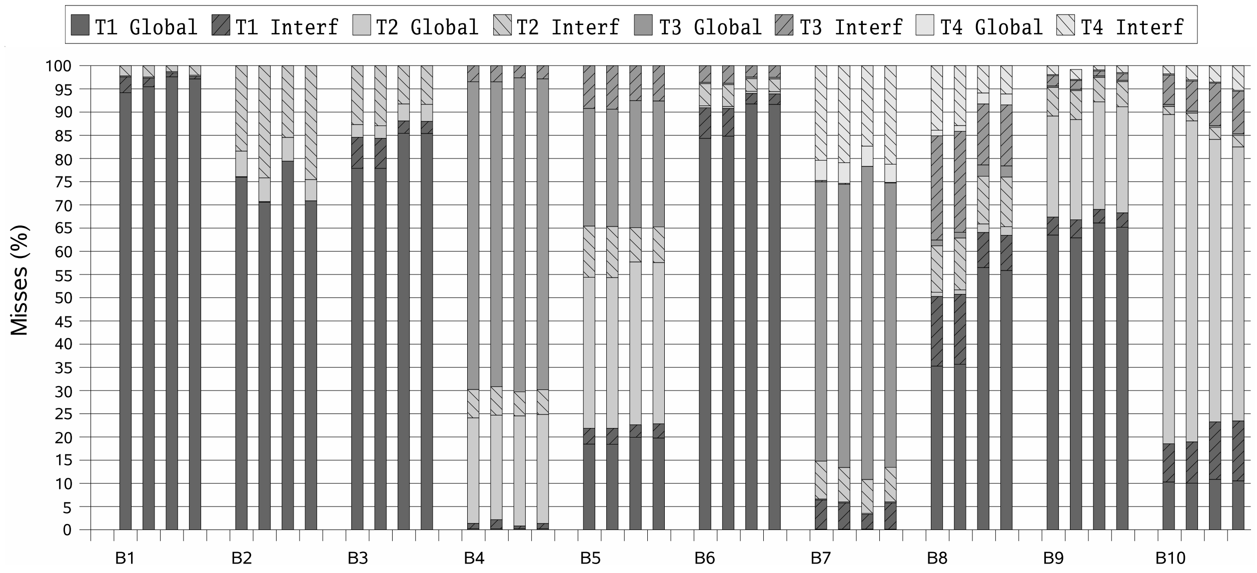

Figure 4: Interference misses for 32KB configurations

requires a larger tag for correctness. 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 [8]. 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 [8] by using dynamic voltage scaling which reduces leakage by a factor 12.

Our study leverages the ideas presented 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. We allow tasks to use any number of associativity ways as long as it is less than or equal to the base line configuration. 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 associativity sets, 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 [27] how this delay can be significantly minimized. It has been shown in [8] that if drowsy cache blocks are not accessed then there is no impact on access time. Since in our approach drowsy lines are never accessed, the cache hit latency is not deteriorated. Drowsy caches increases the cache line by 3% when implemented on a line by line basis. However, our approach does not require this level of granularity.

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). The rectangle size is determined by the cache requirements of the task associated to it. Figure 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 to-

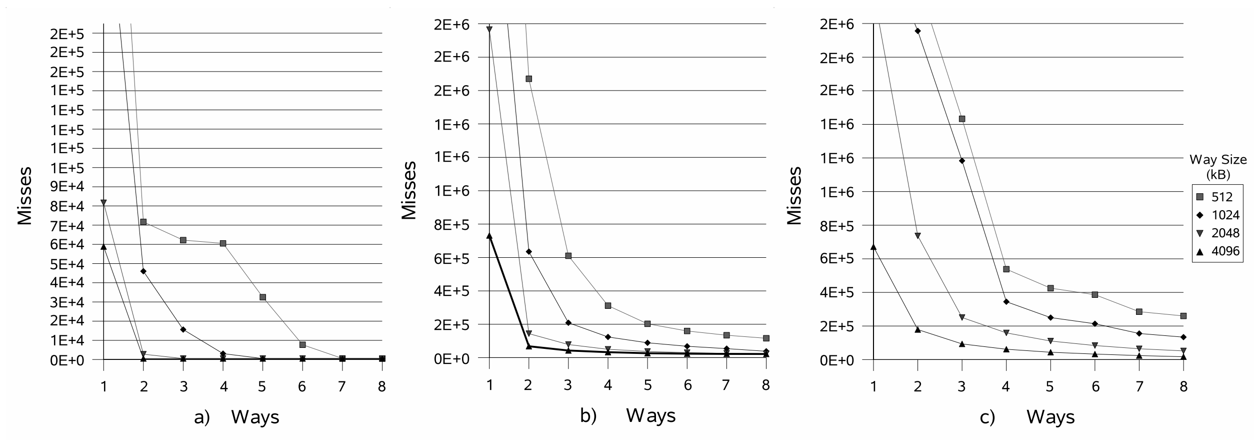

Figure 5: Miss rates for a) ADPCM Decode b) MPEG2 Encode c) Lame Encode

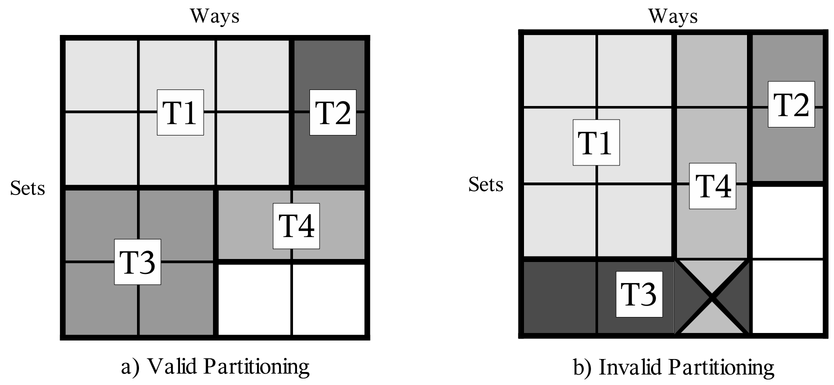

Figure 6: Cache partitioning examples

tal of 4 ways and sets that can be configured down to a quarter of the original sets. Figure 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 further reduction in power. Figure 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. While partitioning for a single task in hardware is feasible [27, 29], 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:

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

where $UTIL(P_i)$ 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:

$$P_i \bigcap P_j = \emptyset, for\ i \neq j$$
$$\sum_{i=1}^{N} COST\left(P_i\right) \leq COST\left(Cache\right)$$
$$VALID(P_i) = TRUE \textit{ for all i}$$

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 *COST(P_i)* refers to the size of the 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 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 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 equal in size while the white circles have no equal partitions. Each partition is simultaneously grown until the tasks utility is greater than $BASE * T$. $BASE$ function is the sum of normal and interference misses from the multitasking profile - it cor-

```
$P = P_i,\ where\ P_i\ is\ partition\ of\ task\ T_i$
$for\ all\ P_i\ =\ Smallest\ Valid\ Partition$
$Set\ tolerance\ value\ T$
$Add\ all\ Pi\ to\ ActiveList$

$while(\ ActiveList\ != \ Empty\ AND\ COST(P)\ <\ COST(Cache))$
    $for(\ Active\ Pi)$
        $if(\ UTIL(P_i)\ >\ BASE(P_i) * T)$
            $Remove\ P_i\ from\ ActiveList$
        $else$
            $GROW(P_i)$
    $if(COST(P))\ >\ COST(Cache))$
    // Partitions can no longer grow
        Option 1: BREAK
        Option 2: Decrease T; Re-Iterate
    if( ActiveList == Empty )
    //Done or Improve Solution
        Option 1: BREAK
        Option 2: Increase T; Re-Iterate
END
```

Figure 7: Heuristic partitioning algorithm

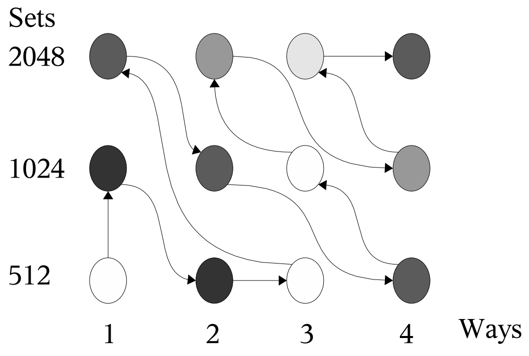

Figure 8: Exploration of partition space

responds 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 the most important tasks. 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

The proposed technique is applied in the following sequence of steps. 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 tasks 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. 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.

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 from 512/1024 to 4096/8192 bytes 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.

5.1 Performance Impact

Figure 9 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

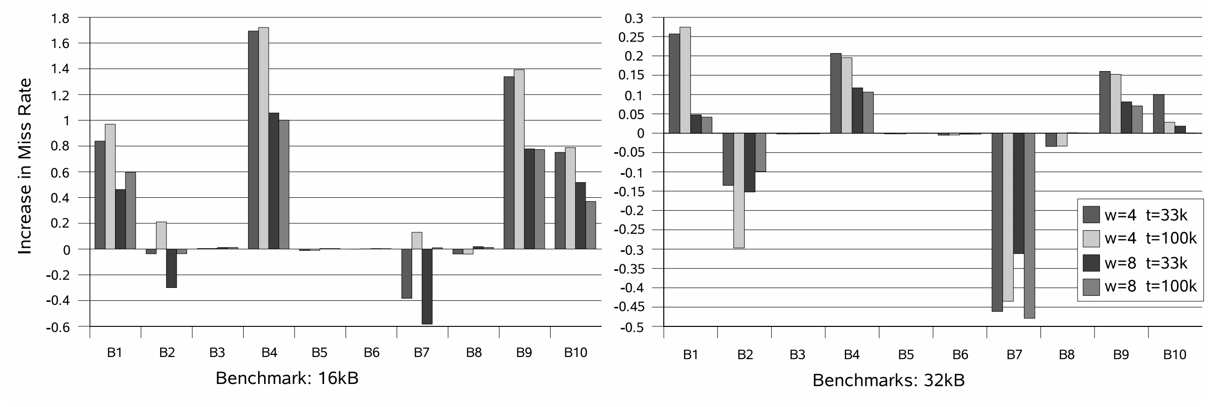

Figure 9: Difference of miss-rate for strict partitioning vs. base configuration

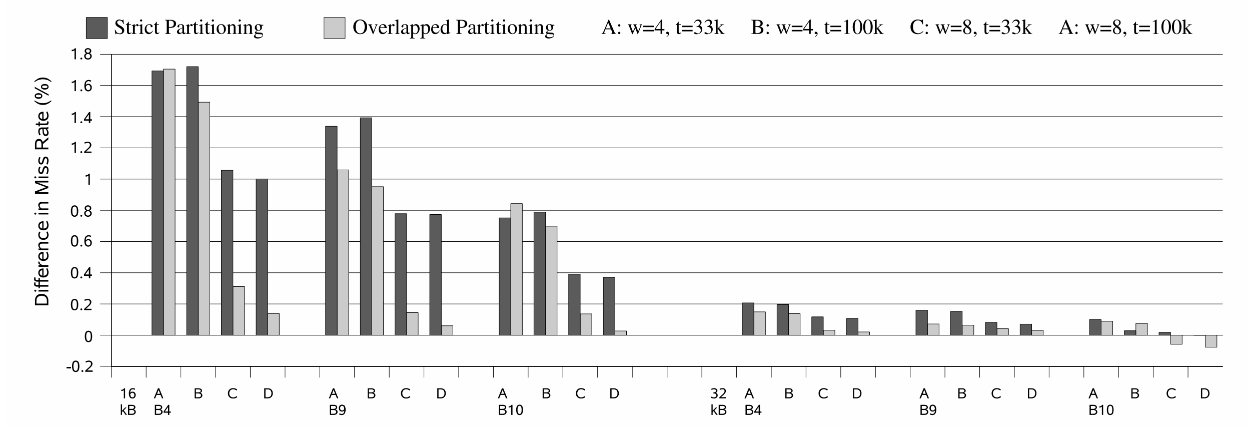

Figure 10: Difference of strict partitioning and overlapped partitioning vs. base configuration

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 then 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, where as if allowed to use the entire cache, it would produce a significant interference.

Figure 10 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 had better performance than simple strict partitioning. This can be attributed to the fact that the non-critical applications do not exhibit strong temporal locality. 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.

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 modeled using Cacti-4.2 [23] with 180nm technology. Each cache partition was modeled as a separate cache and its power characteristics were obtained from Cacti. Caches misses were modeled as accesses to a 256KB direct mapped cache. Figure 11 shows 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%. Eight-way set associative caches showed better improvements over four-way associative caches since the number of ways being accessed is less than 8 for most tasks.

The leakage power reduction is even more significant. In our evaluation of leakage power, we have assumed a drowsy cache implementation as proposed in [8] 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

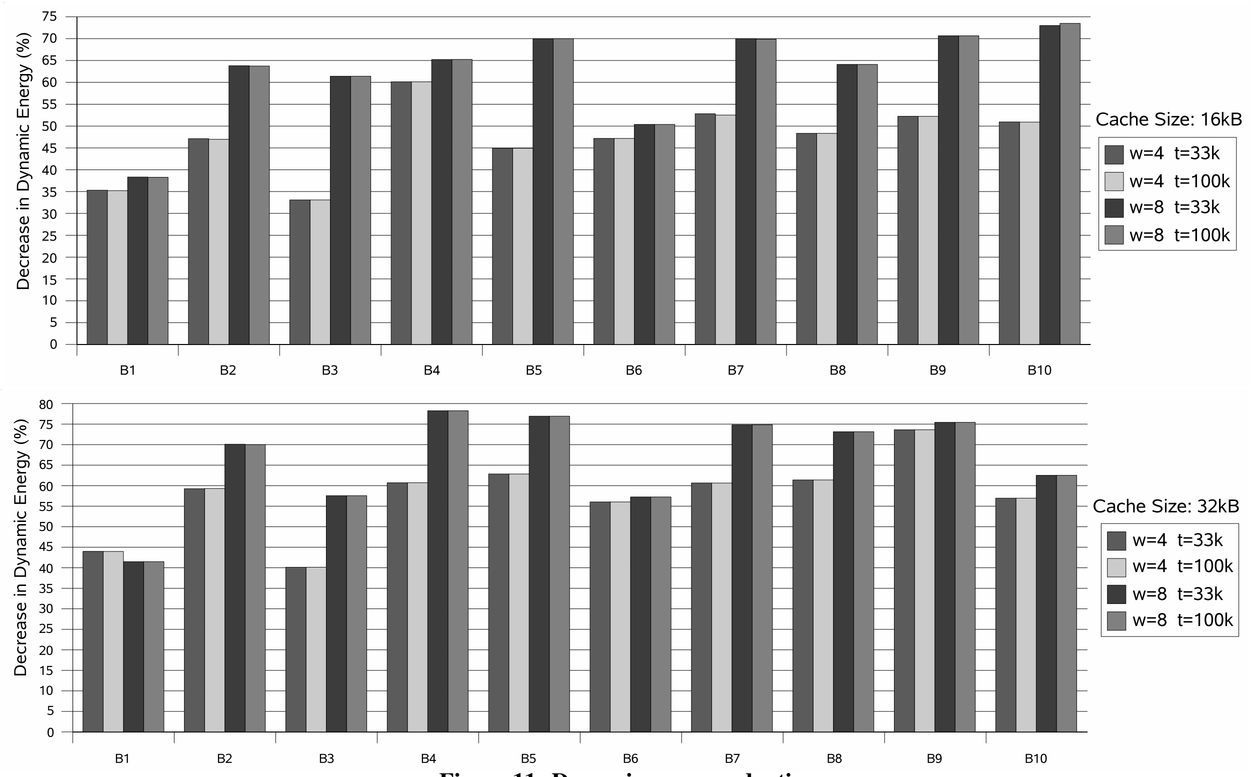

Figure 11: Dynamic power reduction

in drowsy mode and their leakage power reduced by a factor of 12 [8]. Figure 12 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.

6. CONCLUSION

In this paper we have introduced a 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 facilitates the application of multi-tasking support with shared data caches in real-time and energy-efficient embedded systems.

7. REFERENCES

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, 1988.

[2] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *International Symposium on Microarchitecture (MICRO)*, pages 248–259, November 1999.

[3] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behaviour prediction by abstract interpretation. In *Static Analysis Symposium (SAS)*, pages 52–66, 1996.

[4] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *Real-Time Systems Symposium (RTSS)*, page 172âĂŞ181, 1994.

[5] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[6] 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.

[7] H.-C. Chen and J.-S. Chiang. A highly configurable cache architecture for embedded systems. In *Communications, Computers and signal Processing (PACRIM)*, pages 315–318, 2001.

[8] 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)*, pages 148–157, May 2002.

[9] M. Guthaus, J. S. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4: Workshop on Workload Characterization*, pages 3–14, December 2001.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the

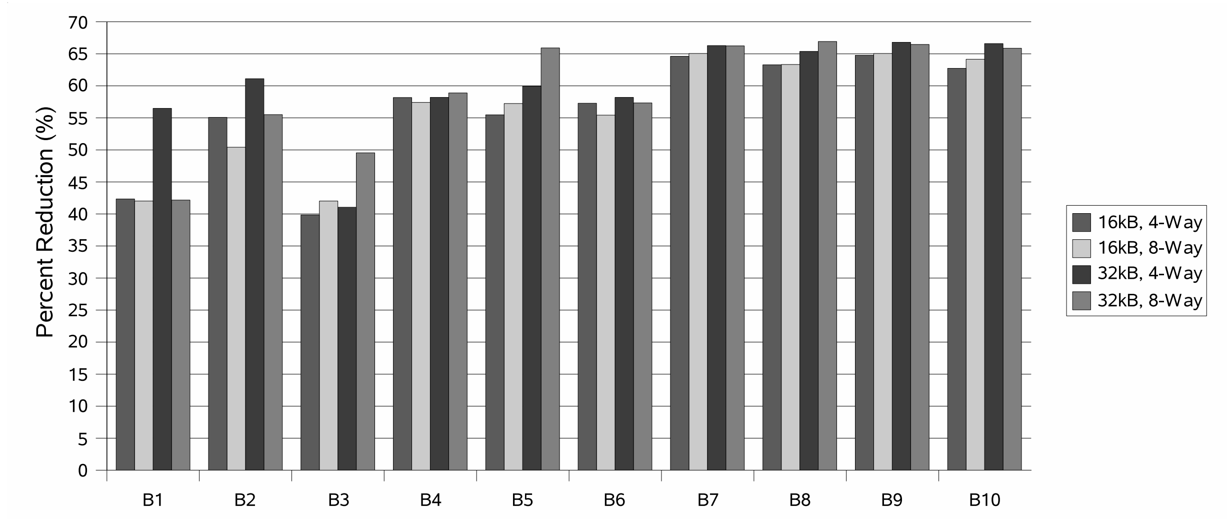

Figure 12: Leakage power reduction

results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, July 2003.

[11] J. Hu, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Analyzing data reuse for cache reconfiguration. *ACM Transactions on Embedded Computing Systems*, 4(4):851–876, 2005.

[12] 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.

[13] 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)*, pages 330–335, December 1997.

[14] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software. In *IEEE Real-Time Systems Symposium,*, pages 148–157, 1997.

[15] 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)*, pages 241–243, 2000.

[16] 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)*, pages 75–84, 1991.

[17] F. Mueller. Compiler support for software-based cache partitioning. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 125–133, 1995.

[18] M. Powell, S.-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)*, pages 90–95, 2000.

[19] M. Powell, S.-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)*, pages 147–157, 2001.

[20] J. Starner and L. Asplund. Measuring the cache interference cost in preemptive real-time systems. In *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 146–154, 2004.

[21] Y. Tan and I. V. J. Mooney. Wcrt analysis for a uniprocessor with a unified prioritized cache. In *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 175–182, 2005.

[22] 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.

[23] 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.

[24] X. Vera, B. Lisper, and X. Jingling. Data caches in multitasking hard real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 145–165, 2003.

[25] S. Wang and L. Wang. Thread-associative memory for multicore and multithreaded computing. In *International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 139–142, 2006.

[26] A. Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering*, 2(3):315–327, 1994.

[27] 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)*, 00:0151, 2002.

[28] C. Zhang, F. Vahid, and R. Lysecky. A self-tuning cache architecture for embedded systems. *ACM Transactions on Embedded Computing Systems*, 3(2):407–425, 2004.

[29] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache architecture for embedded systems. In *International Symposium on Computer Architecture (ISCA)*, pages 136–146, 2003.