Dynamic Cache Reconfiguration for Soft Real-Time Systems

WEIXUN WANG, PRABHAT MISHRA, and ANN GORDON-ROSS, University of Florida

In recent years, efficient dynamic reconfiguration techniques have been widely employed for system optimization. Dynamic cache reconfiguration is a promising approach for reducing energy consumption as well as for improving overall system performance. It is a major challenge to introduce cache reconfiguration into real-time multitasking systems, since dynamic analysis may adversely affect tasks with timing constraints. This article presents a novel approach for implementing cache reconfiguration in soft real-time systems by efficiently leveraging static analysis during runtime to minimize energy while maintaining the same service level. To the best of our knowledge, this is the first attempt to integrate dynamic cache reconfiguration in real-time scheduling techniques. Our experimental results using a wide variety of applications have demonstrated that our approach can significantly reduce the cache energy consumption in soft real-time systems (up to 74%).

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and embedded systems*

General Terms: Design, Performance

Additional Key Words and Phrases: Real-time systems, low-power, embedded systems, cache

ACM Reference Format:
Wang, W., Mishra, P., and Gordon-Ross, A. 2012. Dynamic cache reconfiguration for soft real-time systems. ACM Trans. Embed. Comput. Syst. 11, 2, Article 28 (July 2012), 31 pages.
DOI = 10.1145/2220336.2220340 http://doi.acm.org/10.1145/2220336.2220340

1. INTRODUCTION

Various research efforts in recent years have focused on design and optimization of real-time systems. These systems require unique design considerations due to timing constraints placed on the tasks. Tasks in hard real-time systems must complete execution by their deadlines in order to ensure correct system behavior. Due to these stringent constraints, real-time scheduling algorithms must perform task *schedulability analysis* based on task attributes, such as priorities, periods, and deadlines [Buttazzo 1995; Liu 2000]. A task set is considered *schedulable* if there exists a schedule that satisfies all timing constraints. As embedded systems become ubiquitous, real-time systems with soft timing constraints (missing certain deadlines are acceptable) are gaining widespread acceptance. Soft real-time systems can be found everywhere, including gaming and housekeeping as well as multimedia applications and devices. Tasks in these systems remain effective even if their deadlines are not guaranteed

This work was partially supported by NSF grant CCF-0903430 and SRC grant 2009-HJ-1979.
Authors' address: W. Wang and P. Mishra, Department of Computer and Information Science and Engineering, University of Florida; email: wewang@cise.ufl.edu; A. Gordon-Ross, Department of Electrical and Computer Engineering, University of Florida.

DOI 10.1145/2220336.2220340 http://doi.acm.org/10.1145/2220336.2220340

to be met. Minor deadline misses may result in temporary service or quality degradation but will not lead to incorrect behavior. For example, users of video streaming on mobile devices can tolerate occasional jitters caused by dropped frames, which has a minimal effect on the quality of service and user experience.

One of the most important optimizations in real-time embedded systems is energy consumption reduction, since most of these systems are battery-operated devices. Processor idle time (also known as slack time) provides a unique opportunity to reduce the overall energy consumption by putting the system into sleep mode using dynamic power management (DPM) techniques [Benini et al. 2000]. Alternatively, dynamic voltage scaling (DVS) [Hong et al. 1999] methods can be used to achieve the same goal by reducing the clock frequency such that the tasks execute slowly but do not violate their deadlines [Jejurikar and Gupta 2006; Quan and Hu 2007].

Reconfigurable computing provides the unique ability to tune the system during runtime (*dynamically reconfigure*) to meet optimization goals by changing *tunable* system parameters. The primary aspect of reconfigurable computing research emphasizes tuning algorithms, which determine how and when to dynamically reconfigure tunable parameters to achieve higher performance, lower energy consumption, or balance overall system behavior. One such tunable component is the cache hierarchy. Research has shown that the cache subsystem has become comparable to other components in the processor with respect to the contribution in overall energy consumption [Malik et al. 2000; Segars 2001]. Therefore, since different programs have distinct cache configuration requirements during execution, we can achieve significant energy efficiency as well as performance improvements by employing dynamic cache reconfiguration in the system.

Although reconfigurable caches are highly beneficial in desktop and embedded systems, currently, reconfigurable caches have not been considered in real-time systems due to several fundamental challenges. For example, how to employ and make efficient use of reconfigurable caches in real-time systems remains unsolved. Determining the appropriate cache configuration typically requires some amount of runtime evaluation of different candidate configurations. Furthermore, any change in cache configuration on-the-fly may arbitrarily alter task execution time. In hard real-time systems, the benefit of reconfiguration is limited, since both of these facts can make scheduling decisions difficult and eventually may lead to unpredictable system behavior. However, on the other hand, soft real-time systems offer much more flexibility which can be exploited to achieve considerable energy savings at the cost of minor impacts to the user experience. Our proposed research focuses on real-time systems with soft real-time constraints.

This article presents a novel methodology for using reconfigurable caches in real-time systems with preemptive task scheduling. Our proposed methodology provides an efficient scheduling-aware cache-tuning strategy based on static profiling for both statically and dynamically scheduled real-time systems. Generally speaking, our technique is broadly applicable to any multitasking system. The goal is to optimize energy consumption with performance considerations via reconfigurable cache tuning while ensuring that the majority of the task deadlines are met. In this article, we consider level one (L1) cache reconfiguration only. As shown in Varma et al. [2005], L1 cache energy consumption can play a significant role in overall energy optimization. In fact, many small embedded systems executing light-weight kernels do not have a level two (L2) cache. While the L1 caches we evaluate are small, given that the entire system is also small, L1 caches can still be a significant contributor to overall power consumption. For example, Gordon-Ross et al. [2007] reports a 25% reduction in overall system power by considering L1 cache reconfiguration only. Also, our approach is independent of the actual cache sizes and is applicable as well as beneficial for both larger systems with large L1 caches and smaller systems with small L1 caches. Our follow-up research on dynamic cache reconfiguration for a two-level cache hierarchy in soft real-time systems [Wang and Mishra 2009] considered L2 caches together with L1 caches to achieve overall energy reduction. In that paper, we investigated the interaction between reconfiguration of L1 and L2 caches. The approaches proposed in this article are valid in the context of multi-level cache hierarchy, as shown in Wang and Mishra [2009].

The remainder of this article is organized as follows. Section 2 surveys the background literature addressing both dynamic cache reconfiguration and real-time scheduling techniques. Section 3 describes our proposed research on scheduling-aware cache reconfiguration in soft real-time systems. Section 4 presents our experimental results. Finally, Section 5 concludes this article.

2. RELATED WORK

Nacul and Givargis [2004] proposed an initial work on combining dynamic voltage scheduling and cache reconfiguration on workloads with time constraints. However, their work is applicable in a very restricted scenario where systems do not support task preemption. There are no prior works on dynamic cache reconfiguration in real-time systems that support task preemption. Our proposed research is the first attempt in this direction. This section surveys the background literature in the following three related domains: real-time scheduling techniques, caches in real-time systems, and reconfigurable cache architectures.

2.1 Real-Time Scheduling Techniques

Based on task properties and associated systems, scheduling algorithms can be classified into various types [Liu 2000]. Earliest deadline first (EDF) scheduling [Buttazzo 1995] and rate monotonic (RM) scheduling [Liu 2000] are the most frequently referenced fundamental scheduling algorithms in the real-time systems community. Periodic tasks, which usually have known worst-case execution time (WCET), period, and deadline, are scheduled using such methods. Sporadic tasks are accepted into the system only if the task passes acceptance tests when it arrives. Since sporadic tasks normally have hard time constraints, all accepted tasks are guaranteed to meet their deadlines and are thus treated as periodic tasks. Aperiodic tasks are scheduled whenever enough slack time is available. Hence, aperiodic tasks normally have soft deadlines and can only be scheduled as soon as possible. Scheduling algorithms for tasks with unknown properties, like aperiodic and sporadic tasks, can be found in Liu [2000], Sprunt [1990], and Andersson et al. [2008].

Derived from RM and EDF are energy-aware task scheduling algorithms using energy-optimization techniques and aiming at various objectives, although optimal scheduling has been proved to be an NP-hard problem [Zhang et al. 2007]. DVS and DPM are the most prominent techniques, which exploit variable voltages and power supplies at runtime to reduce energy consumption. Jejurikar and Gupta [2006] addressed the problem in the presence of task mutual controls based on both EDF and RM scheduling. Jejurikar et al. [2004] also proposed in a DVS-enabled scheduling algorithm that is aware of leakage power. Leung et al. [2005] introduced a novel static voltage scheduling algorithm which can result in an energy-optimized slack distribution by relaxing the WCET constraints. Their algorithm compromises average and worst-case execution times of a task to achieve greater energy savings. Quan and Hu [2007] presented a low-complexity voltage scheduling method with fixed priority assignment systems. Since creating additional slack is essential for revealing larger energy savings, Jejurikar and Gupta [2005] deferred task execution in the interest of slack reclamation, further extending low-voltage intervals. Our approach can be used in tandem with any of these state-of-the-art scheduling techniques. In other words, energy-aware scheduling can freely incorporate reconfigurable cache tuning using our methodology to further minimize energy consumption in real-time systems.

2.2 Caches in Real-Time Systems

Cache systems are included in nearly all computing systems to temporarily store frequently accessed instructions and data. Since caches have a much faster access time as compared to that of main memory, caches effectively alleviate the increasing performance disparity between the processor and memory by exploiting the temporal and spatial locality properties of programs. However, historically, incorporating caches into real-time embedded systems faces serious difficulties due to the unpredictability imposed on the system. Caches affect the data access pattern and hence create variations in the data access time. For example, in a preemptive system, since a task may be interrupted by a higher-priority task and resumed again at a later time, the data associated with preempted tasks may be evicted from the cache. This may result in a period of cold-start compulsory cache misses, many of which may have been cache hits if the task had not been preempted. This makes it difficult to calculate a task's worst-case execution time (WCET), which is a prerequisite for most traditional scheduling algorithms.

Since caches introduce intra-task interference so that a specific task's execution time becomes variable at runtime, a great deal of research efforts are directed at employing caches in real-time systems, either by proving schedulability through WCET analysis or avoiding hazardous compulsory miss uncertainty altogether. Cache-aware WCET analysis is a static, design-time analysis of tasks in the presence of caches to predict cache impact on task execution times [Puant 2002]. Cache locking [Puant and Decotigny 2002] is a technique in which useful cache lines are "locked" in the cache when a task is preempted so that these lines will not be evicted to accommodate the new incoming task. Through cache line locking, the WCET and cache behavior becomes more predictable, since the major delay from data replacement and access is avoided. Cache partitioning [Wolfe 1993] is a similar but more aggressive approach in which the cache is partitioned into reserved regions, each of which can only cache data associated with a dedicated task. However, a potential drawback to both cache locking and cache partitioning is per-task reduction of cache resources. To alleviate this limitation, cache-related preemption delay analysis [Tan and Mooney 2007; Staschulat et al. 2005] features tight delay estimation so that prediction accuracy is higher than traditional WCET analysis. This improved accuracy can in turn result in a durable task schedule. Scratch-pad memories, like caches, are also on-chip RAMs but map into the processor's address space at a specified range. Puant and Pais [2007] proposed an offline content-selection algorithm for both scratch-pad memory and caches with line-locking ability to improve both predictability and WCET estimation. Our approach is applicable to real-time systems that employ caches.

2.3 Reconfigurable Cache Architectures

As mentioned in Section 1, in power-constrained embedded systems, nearly half of the overall power consumption is attributed to the cache subsystem [Malik et al. 2000; Segars 2001]. Fortunately, since applications require vastly different cache requirements in terms of cache size, line size, and associativity [Zhang et al. 2004], research shows that specializing the cache to an application's needs can reduce energy consumption by 62% on average [Gordon-Ross and Vahid 2004].

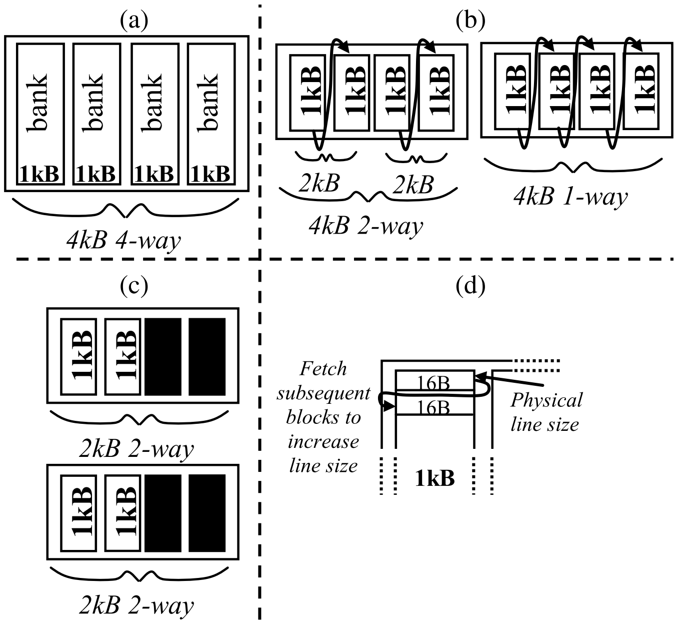


Fig. 1. Cache configurability: (a) base cache bank layout, (b) way concatenation, (c) way shutdown, and (d) configurable line size.

There are many existing general or application-specific reconfigurable cache architectures. Motorola's M*CORE processor [Malik et al. 2000] provided way shutdown and way management, which is the ability to specify the content of each specific way (instruction, data, or unified way). Kim et al. [2000] presented an FPGA-based configurable cache architecture in which part of the cache could serve as a computing unit. Modarressi et al. [2006] developed a cache architecture that could be dynamically partitioned and resized to improve the performance of object-oriented embedded systems. Settle et al. [2006] proposed a dynamically reconfigurable cache specifically designed for chip multiprocessors. The reconfigurable cache architecture proposed by Zhang et al. [2005] imposes no overhead to the critical path, thus cache access time does not increase. Furthermore, the cache tuner consists of a small custom hardware or a lightweight process running on a coprocessor, which can alter the cache configuration via hardware or software configuration registers. The underlying cache architecture consists of four separate banks, as shown in Figure 1(a), each of which acts as a separate way. Way concatenation shown in Figure 1(b), which logically concatenates ways together, enables configurable associativity. Way shutdown shown in Figure 1(c) effectively shuts down ways to vary cache size. Configurable line size in Figure 1(d) is achieved by setting a physical unit-length baseline size and then fetching subsequent physical lines if the logical line size increases.

Given a runtime reconfigurable cache, determining the best cache configuration is a difficult process. Dynamic and static analysis are two possible techniques. With dynamic analysis, cache configurations are evaluated in-system during runtime to determine the best configuration. Two methods are possible for runtime cache analysis. The first method is intrusive and physically changes the cache to each configuration in the design space, examines the effects of each configuration, and chooses the best cache configuration. This method is inappropriate for real-time systems, since it imposes unpredictable performance overhead during exploration. To eliminate this performance overhead, another method employs an N-experts-based analysis [Gordon-Ross et al. 2007]. In this technique, an auxiliary structure evaluates all cache configurations simultaneously. The best cache configuration is determined by inspecting this auxiliary structure, allowing the cache to change to the best configuration in one shot without incurring any performance overhead. Even though this method is nonintrusive, the auxiliary data structure is too power hungry to continuously evaluate the system and thus can only operate periodically.

With static analysis, various cache alternatives are explored, and the best cache configuration is selected for each application in its entirety [Gordon-Ross et al. 2005], that is, application-based tuning, or for each phase of execution within an application [Sherwood et al. 2003], that is, phase-based tuning. Since applications tend to exhibit varying execution behavior throughout their execution, phase-based tuning allows for the cache configuration to be specialized to each particular phase, resulting in greater energy savings than application-based tuning. Regardless of the tuning method, the predetermined best cache configuration (based on design requirements) could be stored in a lookup table or encoded into specialized instructions. The static analysis approach is most appropriate for real-time systems due to its nonintrusive nature. Previous methods focus solely on energy savings or Pareto-optimal points trading off energy consumption and performance. However, none of these methods consider task deadlines, which are imperative in real-time systems. In other words, the existing approaches were designed for desktop and embedded applications but not applicable for real-time systems.

3. SCHEDULING-AWARE CACHE RECONFIGURATION

A major challenge for cache reconfiguration in real-time systems is that tasks are constrained by their deadlines. Even in soft real-time systems, task execution time cannot be unpredictable or prolonged arbitrarily. Our goal is to realize maximum energy savings while ensuring the system only faces an innocuous amount of deadline violations (if any). Our proposed methodology—scheduling-aware cache reconfiguration—provides an efficient and near-optimal strategy for cache tuning based on static program profiling for both statically and dynamically scheduled systems. Our approach statically executes, profiles, and analyzes each task intended to run in the system. The information obtained in the profiling process is fully utilized to make reconfiguration decisions dynamically. The remainder of this section is organized as follows. First, we present an overview of our approach using simple illustrative examples. Next, we present our static analysis technique for cache configuration selection. Finally, we describe how the static analysis results are used during runtime for statically- and dynamically-scheduled real-time systems.

3.1 Overview

This section presents a simple illustrative example to show how reconfigurable caches benefit real-time systems. This example assumes a system with two tasks, *T1* and *T2*. Traditionally if a reconfigurable cache technique is not applied, the system will use a *base cache* configuration Cache_{base}, which is defined in Definition 3.1.

Definition 3.1 (*Base Cache*). The term refers to the configuration selected as the optimal cache for tasks in the target system with respect to energy, as well as performance, based on static analysis. Caches in such systems are chosen to ensure durable task schedules, and their configurations are fixed throughout all task executions.

In the presence of a reconfigurable cache, as shown in Figure 2, different optimal cache configurations are determined for every *phase* of each task. For ease of

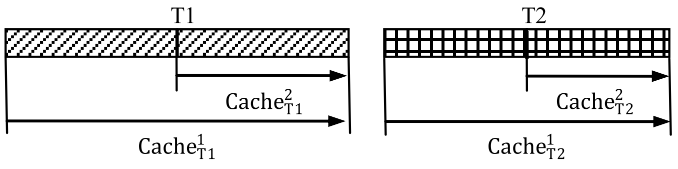


Fig. 2. Cache configurations selected based on task phases.

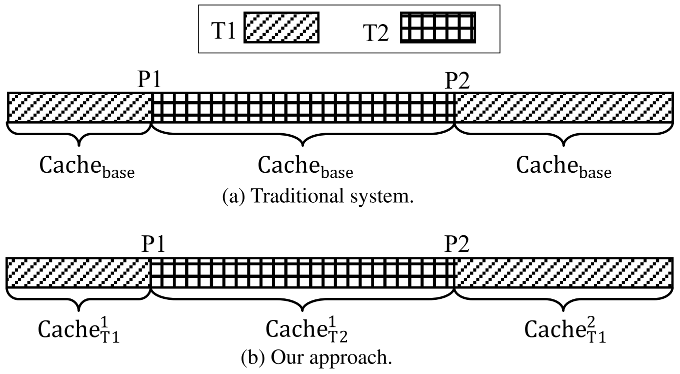


Fig. 3. Dynamic cache reconfiguration for tasks *T1* and *T2*.

illustration, we divide each task into two phases: $phase_1$ spans the entire execution from beginning to end, and $phase_2$ spans from the half way point of the dynamic instruction flow (midpoint) to the end of execution. The terms Cache^1_{T1}, Cache^2_{T1}, Cache^1_{T2}, and Cache^2_{T2} represent the optimal cache configurations for $phase_1$ and $phase_2$ of task *T1* and *T2*, respectively. These configurations are chosen statically to be more energy efficient (with the same or better performance), in their specific phases, than the global base cache, Cache_{base}.

Figure 3 illustrates how energy consumption can be reduced by using our approach in real-time systems. Figure 3(a) depicts a traditional system, and Figure 3(b) depicts a system with a reconfigurable cache (our approach). In this example, *T2* arrives (at time *P1*) and preempts *T1*. In a traditional approach, the system executes using Cache_{base} exclusively. With a reconfigurable cache, the first part of *T1* executes using Cache^1_{T1}. Similarly, Cache^1_{T2} is used for execution of *T2*. Note that the actual preemption point of *T1* is not exactly at the same place where we precomputed the optimal cache configuration (midpoint), since tasks may arrive at any time. When *T1* resumes at time point *P2*, the cache is tuned to Cache^2_{T1} since the actual preemption point is closer to the midpoint as compared to the starting point. The overall energy consumed using a reconfigurable cache results in energy savings due to the use of different energy-optimal caches for each phase of task execution as compared to using one global base cache in the traditional system. Our experimental results suggest that the proposed approach can significantly reduce energy consumption of the memory subsystem with very little performance penalty.

3.2 Phase-Based Optimal Cache Selection

This section describes our static analysis approach for determining the optimal cache configurations for various task phases. In a preemptive system, tasks may be interrupted and resumed at any point in time. Each time a task resumes, cache performance for the remainder of task execution will differ from the cache performance for

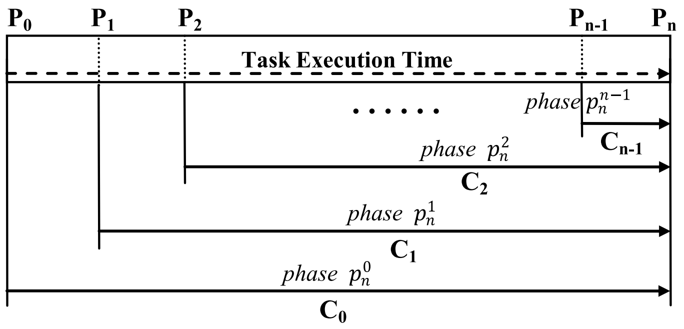


Fig. 4. Task partitioning at n potential preemption points (P_i) resulting in n phases. Each phase comprises execution from the invocation/resumption point to task completion. C_i denotes the cache configuration used in each phase.

the entire application due to its own distinguishing behaviors as well as cold-start compulsory cache misses. Therefore, the optimal cache configuration for the remainder of the task execution may be different.

Definition 3.2 (*Phase*). Phase is defined as the execution period between one potential preemption point (also called a *partition point*) and task completion. The phase that starts at the ith partition point is denoted as phase p_n^i, where n is the total number of phases of that task.

Figure 4 depicts the general case in which a task is divided by $n - 1$ predefined *potential preemption points* ($P_1, P_2 \dots P_{n-1}$). P_0 and P_n are used to refer to the start and end points of the task, respectively. Here, C_0, C_1 ... C_{n-1} represent the optimal cache configuration (either energy or performance) for each phase, respectively. To observe the variation in cache requirements for each phase, Table I shows the variation in energy-optimal and performance-optimal instruction and data caches for each phase. For example, the energy-optimal cache configuration for the phase starting from the midpoint to completion (C_2) of benchmark *cjpeg* has a 2048-byte capacity, a 16-byte line size, and two-way associativity.

During static profiling, a *partition factor* is chosen that determines the number of potential preemption points and resulting phases. Partition granularity is defined as the number of dynamic instructions between two partition points and is determined by dividing the total number of dynamically executed instructions by the partition factor. Intuitively, the optimal partition granularity should be a single instruction, potentially leading to the largest amount of energy savings. However, such a tiny granularity would result in a prohibitively large look-up table, which is not feasible due to area as well as searching time constraints. Due to cache locality over time, the optimal performance cache tends to be the largest cache [Hennessy and Patterson 2003], and the optimal energy cache is not necessarily the smallest dynamic energy cache [Gordon-Ross and Vahid 2004]. Thus, a trade-off should be made to determine a reasonable partition factor based on energy-savings potential and acceptable overheads. An important question one can raise is whether a larger partition factor (finer granularity) always reveals more energy savings. However, to answer this question, we need to address the following two issues.

The first issue is how the optimal cache configuration for each phase varies when the partition factor increases. We noticed that for each task, once the partition factor is larger than a certain threshold, more and more neighboring partitions share the same optimal cache configuration. We explored how the partition factor could affect the variation of optimal (both energy and performance) cache configurations for

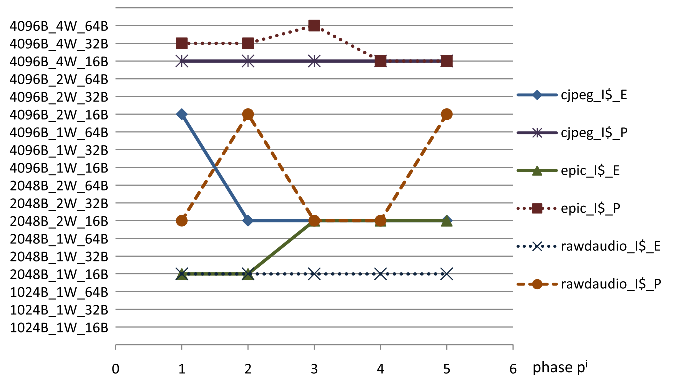
Table I. Optimal Cache Configurations for Task Phases

	CJPEG			
	I-Cache		**D-Cache**	
	Energy Optimal	**Performance Optimal**	**Energy Optimal**	**Performance Optimal**
C0	4KB_2W_16B	4KB_4W_16B	4KB_4W_16B	4KB_4W_16B
C1	4KB_2W_16B	4KB_4W_32B	4KB_4W_16B	4KB_4W_16B
C2	2KB_2W_16B	4KB_4W_16B	2KB_2W_32B	4KB_4W_16B
C3	2KB_2W_16B	4KB_4W_16B	2KB_2W_32B	4KB_4W_16B
	RAWCAUDIO			
	I-Cache		**D-Cache**	
	Energy Optimal	**Performance Optimal**	**Energy Optimal**	**Performance Optimal**
C0	1KB_1W_16B	4KB_2W_64B	2KB_2W_16B	2KB_2W_16B
C1	1KB_1W_16B	2KB_2W_16B	2KB_2W_16B	4KB_4W_16B
C2	1KB_1W_16B	4KB_4W_16B	2KB_2W_16B	4KB_4W_16B
C3	1KB_1W_16B	4KB_2W_16B	2KB_2W_32B	4KB_4W_16B
	A2TIME01			
	I-Cache		**D-Cache**	
	Energy Optimal	**Performance Optimal**	**Energy Optimal**	**Performance Optimal**
C0	4KB_4W_16B	4KB_4W_16B	4KB_2W_32B	4KB_4W_16B
C1	4KB_4W_16B	4KB_4W_16B	2KB_2W_32B	4KB_4W_16B
C2	4KB_4W_16B	4KB_4W_16B	2KB_2W_16B	4KB_4W_16B
C3	4KB_4W_16B	4KB_4W_16B	2KB_2W_16B	2KB_2W_16B

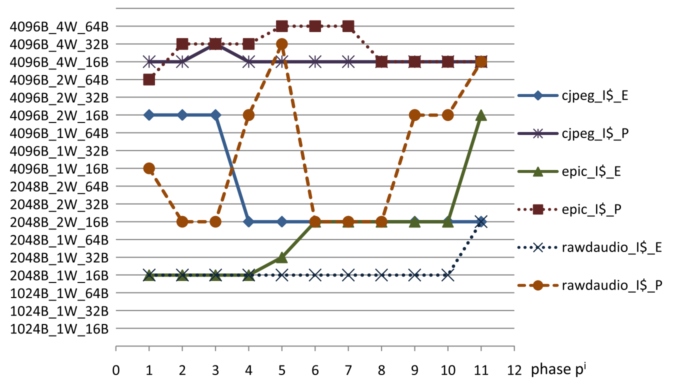
Note: Each configuration is denoted by the total cache size in kilobytes (KB), followed by the associativity in number of ways (W), followed by the line size in bytes (B).

each benchmark in MediaBench [Lee et al. 1997] and EEMBC [EEMBC 2000]—the two benchmark suites we use in Section 4. Figure 5 shows the results for some of the benchmarks (*cjpeg*, *epic*, and *rawdaudio*) using partition factors 6, 12, and 18. For the same benchmark, the optimal cache configuration for each phase varies in a consistent pattern across different partition factors. For example, the energy-optimal instruction cache configuration for benchmark *cjpeg* (cjpeg_I$_E) is 4096B_2W_16B[1] for the first several phases and then changes to 2048B_2W_16B starting from approximately one third of the program: *phase* p_6^2, *phase* p_{12}^4, and *phase* p_{18}^6 when the partition factor is 6, 12, and 18, respectively. In other words, a larger partition factor makes more and more phases share the same optimal cache configuration with their neighboring phases. Different optimal cache configurations can be found when the partition factor increases. For example, the performance-optimal instruction cache configuration for benchmark *cjpeg* (cjpeg_I$_P in Figure 5(b)) with partition factor 12 differs at *phase* p_{12}^3, compared to the similar position when the partition factor is 6 (Figure 5(a)). Our experimental results show that though discrepancies do happen, their impact on energy savings is normally negligible, because the energy/performance difference between the newly selected cache configuration and the original one is usually very small. From this observation, one can derive the fact that application behavior can sufficiently be captured by a certain partition factor. This is evident due to the well-established 90/10 rule of

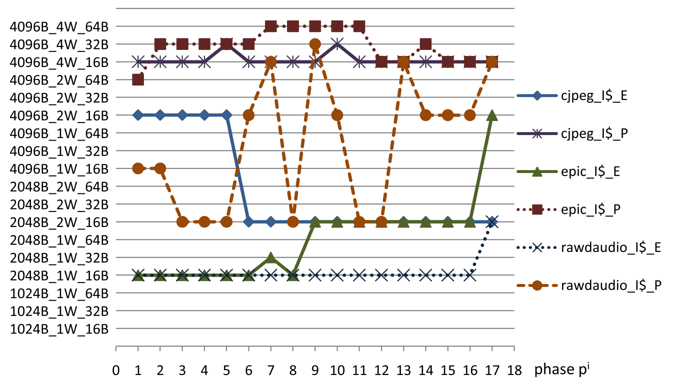
[1]A cache configuration with a 4096-byte capacity, 16-byte line size, and two-way associativity.



(a) Partition factor = 6.



(b) Partition factor = 12.



(c) Partition factor = 18.

Fig. 5. Optimal cache configuration variation under different partition factors. (In this figure, I$ represents the instruction cache, 'E' stands for energy-optimal, and 'P' stands for performance-optimal.)

execution—90% of the execution time is spent in only 10% of the code—in which the 90% of the time is typically spent executing loops. For each loop iteration, except the first and last iterations, execution behavior is typically similar, thus resulting in the same optimal cache configuration for all other iterations. For a loop with N iterations,

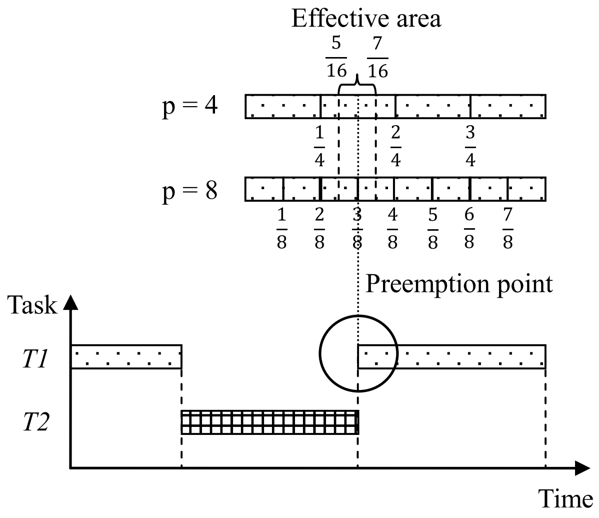


Fig. 6. Effective range in which a higher partition factor makes a difference.

the partition factor only needs to be large enough to capture all dynamic instructions of iterations two through $(N-1)$, as any smaller granularity would capture a subset of iterations, each of which may have the same optimal configuration. Thus, we define a *stage* of execution as a range of consecutive dynamic instructions in which a common optimal cache configuration exists.

The second issue is whether a finer partition granularity always results in more energy savings than a coarser granularity. With a finer granularity, if there is no extra variation in the optimal cache configuration across phases, there will be no additional energy savings, since the same cache configurations are being used. If variations can be observed, according to our experiments, they only happen at stage boundaries, which is a very limited portion in the entire program. Figure 6 gives an example explaining why this is the case. Suppose there are two tasks *T1* and *T2* in the system, and the partition factor (p) can be chosen as four or eight. A valid schedule for these tasks is shown in Figure 6. Since *T2* is executed as a whole, the cache configuration used is the optimal one for the entire task, which is the same as both partition factors. *T1* is preempted by *T2*. When *T1* resumes, a different cache configuration should be picked based on the preemption point as well as the partition factor. As discussed in the first issue, a higher partition factor shows a consistent variation pattern in optimal cache configuration with only minor exceptions. Suppose when the partition factor is four, for task *T1*, the cache configuration picked for *phase* p_4^0, *phase* p_4^1, *phase* p_4^2, and *phase* p_4^3 are C_A, C_A, C_B, and C_B, respectively. And when partition factor is eight, they are C_A, C_A, C_A, C_C, C_B, C_B, C_B, and C_B for *phase* p_8^0 to *phase* p_8^7, respectively. Using the nearest-neighbor technique, as discussed in Section 3.4.1, the advantage of using a partition factor of eight over four becomes effective only when the preemption happens within the range from 5/16 to 7/16 of *T1* (*effective area*), since C_C will be chosen instead of C_A or C_B. Note that C_C may be more energy/performance efficient for the remaining part of *T1* than C_A and C_B. From the entire system's point of view, a higher partition factor (i.e., eight) does not help for *T2* as well as *T1* if the application does not get preempted or the preemption does not happen in the *effective area*. Based on our experiments, energy savings increases by 3–8% (for that one task) only when the preemption occurs within the effective area of the dynamic instruction flow. Empirically, the effective area is typically 5–8% of the task. Due to these two small probabilities multiplied together, 0.4% on average, a finer-granularity partition can realize only minor benefits.

Thus, the goal of a system designer is to find a partition factor that maximizes the energy reduction and minimizes the number of partition points that need to be stored. The rule of thumb is to find a partition factor minimizing the number of neighboring partitions that share the same optimal cache configuration. It could be a local optimal factor for each task if a varying number of table entries for different tasks is allowed, or it could be a global optimal factor for the task set. Based on our experience, a partition factor ranging from four to seven is sufficient to make our technique work efficiently.

Static profiling generates a *profile table* that stores the potential preemption points and the corresponding optimal cache configurations for each task. Sections 3.3 and 3.4 describe how this profile table is used during runtime in statically and dynamically scheduled systems.

3.3 Statically Scheduled Systems

With static scheduling, arrival times, execution times, and deadlines are known a priori for each task, and this information serves as scheduler input. The scheduler then provides a schedule detailing all actions taken during system execution. According to this schedule, we can statically execute and record the energy-optimal cache configurations that do not violate any task's deadline for every execution period of each task. For soft real-time systems, global (system-wide) energy-optimal configurations can be selected, as long as the configuration performance does not severely affect system behavior. After this profiling step, the profile table is integrated with the scheduler so that the cache reconfiguration hardware (cache tuner) can tune the cache for each scheduling decision.

3.4 Dynamically Scheduled Systems

With dynamic scheduling (online scheduling), scheduling decisions are made during runtime. In this scenario, task preemption points are unknown, since new tasks may enter the system at any time with any time constraint. In this section, we present two versions of our technique based on the target system's characteristics.

3.4.1 Conservative Approach. In some soft real-time systems where high service quality is required and time constraints are pressing, only an extremely small number of violations are tolerable. The conservative approach could ensure that given a carefully chosen partition factor, almost every task could meet their deadlines with only a few exceptions. To ensure the largest task schedulability, any reconfiguration decision will only change the cache into a lowest energy configuration whose execution time is not longer than that of the base cache. In other words, to maintain a high quality of service, only cache configurations with equal or higher performance than the base cache are chosen for each task phase. Note that the chosen energy-optimal configuration may not be the global lowest energy configuration but is the configuration with the lowest energy consumption given a specific time constraint. We denote them as deadline-aware energy-optimal cache configurations.

The scheduler chooses the appropriate cache configuration from the generated profile table that contains the energy-optimal cache configurations for each task phase. Table II(a) shows the profile table for task i with a partition factor p. $EO_i(n/p)$ represents the energy-optimal cache configuration for *phase* p_p^n of task i. Here, n/p represents the nth phase out of p phases. The total dynamic instruction count (TIN) refers to the number of dynamic instructions executed in a single run of that task.

Table II(a). Static Profile Table for the Conservative Approach

Task ID: i	**Partition Factor: p**
Total Instruction Number (TIN)	
EO_i(0/p)	
EO_i(1/p)	
EO_i(2/p)	
......	
EO_i(p-1/p)	

Table II(b). Task List Entry for Task *i* for the Conservative Approach

Task ID: i	**Partition Factor: p**
Arrival time (A_i)	**Deadline (D_i)**
Total Instruction Number (TIN)	**Executed Instruction Number (EIN)**
EO_i(0/p)	
EO_i(1/p)	
EO_i(2/p)	
......	
EO_i(p-1/p)	

During system execution, the scheduler maintains a task list keeping track of all existing tasks, as shown in Table II(b). In addition to the static profile table of Table II(a), runtime information, such as arrival time (A_i), deadline (D_i), and the number of already executed dynamic instructions (EIN), are also recorded. This information is stored not only for the scheduler but also for the cache tuner. When a newly arrived task[2] begins execution for the first time, the deadline-aware energy-optimal cache configuration $EO_i(0/p)$ is obtained from the task list entry, and the cache tuner adjusts the cache appropriately. If preemption happens, the number of the preempted task's executed instructions (EIN) is calculated and stored in its task list entry.

As indicated in Section 3.2, potential preemption points are predetermined during the profile table generation process. However, it is highly unlikely that the actual preemptions would occur precisely on these potential preemption points. Hence, a *nearest-neighbor* method is used to determine which cache configuration should be used. Essentially, if the preemption point falls between partition points n/p and $(n + 1)/p$, the nearest point would be used to select the current cache configuration. Algorithm 1 illustrates the cache-tuning algorithm for our conservative approach. This algorithm is called when a previously preempted task resumes its execution. It runs in a time complexity of $O(p)$, where p is the partition factor. Note that the returned cache configuration information is sent to the cache tuner.

As our experimental results show, the conservative approach obtains significant energy savings with little or no impact on the quality of service. A minor number of time-constraint violations are caused by cache behaviors in which the optimal cache

[2]To be more specific, we actually mean "jobs" (execution instances of tasks). For simplicity, a more general term "task" is used in this article.

ALGORITHM 1: Cache Configuration Selection for a Resumed Preempted Task in the Conservative Approach

```
Input: Task list entry
Output: A deadline-aware cache configuration for the resumed task Tc.
for $i = 0$ to $p - 2$ do
  if $TIN_{Tc} \times i/p \leq EIN_{Tc} < TIN_{Tc} \times (i+1)/p$ then
    if $(EIN_{Tc} - TIN_{Tc} \times i/p) < (TIN_{Tc} \times (i+1)/p - EIN_{Tc})$ then
      $PHASE_{Tc} = i/p$;
    else
      $PHASE_{Tc} = (i+1)/p$;
    end if
  end if
end for
if $EIN_{Tc} \geq TIN_{Tc} \times (p-1)/p$ then
  $PHASE_{Tc} = (p-1)/p$;
end if
$Cache_{Tc} = EO_i(PHASE_{Tc})$;
Return: $Cache_{Tc}$
```

configuration for the period from one preemption point to another preemption point and for the pre-decided phase differ greatly. In other words, the chosen cache configuration may happen to be inefficient for the execution period between two actual preemption points such that the lost time is not reparable by the subsequently selected cache configurations in that task. Fortunately, this kind of behavior is relatively rare.

3.4.2 Aggressive Approach. For soft real-time systems in which only moderate service quality is needed, a more aggressive version of our approach can reveal additional energy savings at the cost of possibly violating several future task deadlines, but the number of missed deadlines would remain in an acceptable range.

Similar to the conservative approach, a profile table is associated with every task in the system; however, this profile table contains the performance-optimal cache configuration (whose execution time is the shortest) in addition to the energy-optimal configuration (the configuration with the lowest energy consumption among all of the candidates) for every task phase. In order to assist dynamic scheduling, the profile table also includes the corresponding phase's execution time (in cycles) for each configuration. Table III(a) shows the profile table for task i with a partition factor of p. The terms *EO*, *EOT*, *PO*, and *POT* stand for the energy-optimal cache configuration, the energy-optimal cache configuration's execution time, the performance-optimal cache configuration, and the performance-optimal cache configuration's execution time, respectively. Notice that the performance and energy efficiency of a cache configuration is not inversely proportional. The energy-optimal configuration does not necessarily have the worst performance. Compared to the base cache, the energy-optimal configuration could have both better energy efficiency and performance.

Table III(b) shows the task list entry for the aggressive approach. The difference from the conservative approach (shown in Table II(b)) is that every task list entry also holds a current phase identifier (CP_i). CP_i denotes the partition point that this task's execution just passed and is useful for cache reconfiguration upon task resumption. Note that a newly inserted task's *CP* is initialized to 0. In addition to the task list, the scheduler also maintains another runtime data structure called the ready task list

Table III(a). Static Profile Table for the Aggressive Approach

Task ID: i		Partition Factor: p	
Total Instruction Number (TIN)			
$EO_i(0/p)$	$EOT_i(0/p)$	$PO_i(0/p)$	$POT_i(0/p)$
$EO_i(1/p)$	$EOT_i(1/p)$	$PO_i(1/p)$	$POT_i(1/p)$
$EO_i(2/p)$	$EOT_i(2/p)$	$PO_i(2/p)$	$POT_i(2/p)$
......			
$EO_i(p\text{-}1/p)$	$EOT_i(p\text{-}1/p)$	$PO_i(p\text{-}1/p)$	$POT_i(p\text{-}1/p)$

Table III(b). Task List Entry for Task *i* for the Aggressive Approach

Task ID: i		Partition Factor: p	
Arrival time (A_i)		Deadline (D_i)	
Total Instruction Number (TIN)		Executed Instruction Number (EIN)	
Current Phase (CP)			
$EO_i(0/p)$	$EOT_i(0/p)$	$PO_i(0/p)$	$POT_i(0/p)$
$EO_i(1/p)$	$EOT_i(1/p)$	$PO_i(1/p)$	$POT_i(1/p)$
$EO_i(2/p)$	$EOT_i(2/p)$	$PO_i(2/p)$	$POT_i(2/p)$
......			
$EO_i(p\text{-}1/p)$	$EOT_i(p\text{-}1/p)$	$PO_i(p\text{-}1/p)$	$POT_i(p\text{-}1/p)$

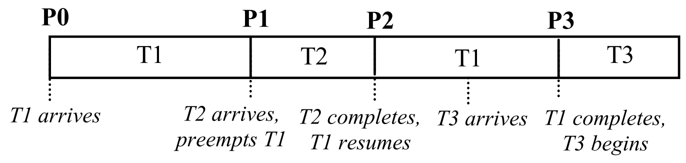


Fig. 7. Task set and sample scheduling.

(RTL), which contains an identifier for each existing task currently ready to execute in the system.

To explain the aggressive approach, we use an illustrative example in which there are three tasks (jobs), *T1*, *T2*, and *T3*, with absolute deadlines D_{T1}, D_{T2}, and D_{T3}, where $D_{T2} < D_{T1} < D_{T3}$. According to EDF, the priority sequence is simply the opposite of the deadlines, which is $Pri_2 > Pri_1 > Pri_3$. Figure 7 shows a schedule for these tasks. Note that *P0*, *P1*, *P2*, and *P3* represent the time instances when any event (arrival, completion, etc.) occurs. At time point *P0*, *T1* arrives and the scheduler generates the task list entry for *T1* and adds *T1* to the *RTL*. Since *T1* is currently the only task in the system, the scheduler instructs the cache tuner to configure the cache to $EO_{T1}(0/p)$ if and only if $P0 + EO_{T1}(0/p) < D_{T1}$, otherwise the cache will be tuned to $PO_{T1}(0/p)$, which ensures that *T1*'s deadline will be met. At time point *P1*, *T2* arrives with priority higher than the currently active task *T1*. The scheduler calculates *T1*'s current phase CP_{T1} and updates *T1*'s task list entry. Note that *T1*'s deadline may be violated if the following inequality holds.

$$P1 + POT_{T1}((CP_{T1} + 1)/p) + POT_{T2}(0/p) > D_{T1}. \tag{1}$$

This is obviously an underestimation of the execution time that the remaining portion of *T1* would take and thus more aggressive, but it favors tasks with higher priority (*T2*). However, if we use $POT_{T1}(CP_{T1}/p)$ in Equation (1), *T2* may have a lower chance of being accepted, but the lower priority task *T1* would more likely meet its deadline.

If Equation (1) does not hold, the scheduler determines *T2*'s cache configuration C_{T2} as follows (assuming $P_i + POT_i(0/p) < D_i$ for all tasks i, otherwise task i is not schedulable in any situation).

```
if ($P1 + EOT_{T2}(0/p) > D_{T2}$) then
    $C_{T2} = PO_{T2}(0/p)$;
else if ($P1 + EOT_{T2}(0/p) + POT_{T1}((CP_{T1} + 1)/p) < D_{T1}$) then
    $C_{T2} = EO_{T2}(0/p)$;
else if ( $P1 + EOT_{T2}(0/p) + POT_{T1}((CP_{T1} + 1)/p) > D_{T1}$) then
    $C_{T2} = PO_{T2}(0/p)$.
```

At time point *P2*, *T2* completes and *T1* resumes, since it is the only ready task. The scheduler utilizes CP_{T1} to determine the appropriate partition for choosing a cache configuration. This technique is similar in principle to the nearest-neighbor method used in Section 3.4.1 except that a decision should be made whether to use the energy-optimal or performance-optimal configuration based on the remaining time budget. At some point during *T1*'s execution, *T3* arrives, but since *T3* has a lower priority than *T1*, *T3* begins execution after *T1* completes execution. By this time, *T3* is the only task, and its cache configuration decision is made using the same method as task *T1* at time *P0*.

Algorithm 2 illustrates the general cache configuration selection algorithm for preempted tasks for our aggressive approach. This algorithm is called either when a new task with a higher priority than the current executing task arrives or when the current task finishes execution. In the former case, Step 1 uses the executed instruction number (EIN) to calculate the current phase (CP) for the preempted task. While in the latter case, this step should be omitted. Step 2 selects the highest priority[3] task Tc from RTL. In the former case, the newly arrived task is inserted into RTL and, obviously, Tc refers to that task. Step 3 checks the schedulability of all of the tasks in RTL by iteratively checking whether each task can meet its deadline if all the preceding tasks, including itself, use performance-optimal cache configurations. This process is done in decreasing order of task priority to achieve the smallest number of discarded tasks. In Step 4, the appropriate cache configuration for Tc is selected based on whether it is safe to use the energy-optimal cache configuration. This algorithm runs in time of O(max(p,m)), where p is the partition factor and m is the total number of tasks in RTL.

3.5 Impact of Storing Multiple Cache Configurations

This section investigates the extent to which individual cache configuration candidates are required during scheduling. In the approaches proposed in Sections 3.4.1 and 3.4.2, the scheduler only considers either the energy-optimal cache in the conservative approach or the energy- and performance-optimal caches in the aggressive approach, for each task phase. As justified by our experiments, we can achieve a considerable amount of energy savings at the cost of very low system overheads simply by storing these cache configurations in the static profile table. However, there exists other configurations that offer Pareto-optimal trade-off points. Simply because the energy-optimal cache cannot satisfy a particular task's deadline, it does not mean that there is no cache configuration for that task that can meet the deadline and consume less

[3]Here, priority means the dynamic scheduling priority decided using EDF.

ALGORITHM 2: Cache Configuration Selection for the Aggressive Approach

```
Input: Task list entry, ready task list and preemption point
Output: A appropriate cache configuration
Step 1: Calculate CP for the preempted task Tp. Insert Tp to RTL.
for $i = 0$ to $p - 1$ do
  if $TIN_{Tp} \times i/p \leq EIN_{Tp} < TIN_{Tp} \times (i+1)/p$ then
    $CP_{Tp} = i/p$;
  end if
end for
Step 2: Remove the task with maximum priority Tc from RTL.
Step 3: Sort all tasks in RTL by priority, $T_1$ to $T_m$, from highest to lowest. C represents the
current time instant.
for $j = 1$ to $m$ do
  if $C + POT_{Tc}(CP_{Tc}/p) + \sum_{i=1}^{j} POT_{Ti}((CP_{Ti}+1)/p) > D_{Tj}$ then
    Task $D_{Tj}$ is subject to be discarded;
  end if
end for
Step 4: Select cache configuration for Tc. Let m′ be the number of tasks in RTL left after
Step 3.
if $C + EOT_{Tc}(CP_{Tc}/p) > D_{Tc}$ then
  $Cache_{Tc} = PO_{Tc}$;
else
  $EO\_OK = true$;
  for $j = 1$ to $m'$ do
    if $C + EOT_{Tc}(CP_{Tc}/p) + \sum_{i=1}^{j} POT_{Ti}((CP_{Ti}+1)/p) > D_{Tj}$ then
      $EO\_OK = false$;
    end if
  end for
end if
if $EO\_OK == true$ then
  $Cache_{Tc} = EO_{Tc}$;
else
  $Cache_{Tc} = PO_{Tc}$;
end if
Return: $Cache_{Tc}$
```

energy than the performance-optimal cache. For example, as described in Algorithm 2, when the scheduler finds that using the energy-optimal cache for a task is unsafe, it has no choice but to select the performance-optimal cache, but if the second energy-optimal cache is also available to the scheduler and is able to meet the time constraint (has higher performance), the scheduler can select that cache configuration to potentially save more energy. Figure 8(a) illustrates this extension of the profile table.

Note that we use the phrase *second beneficial energy-optimal cache* and *second beneficial performance-optimal cache* in Figure 8(a). Figure 8(b) shows how we select these caches. We only consider those cache configurations on the Pareto-optimal curve that have either better energy efficiency or higher performance than the other configurations. In the extreme case, if we can store all these cache configurations for every task

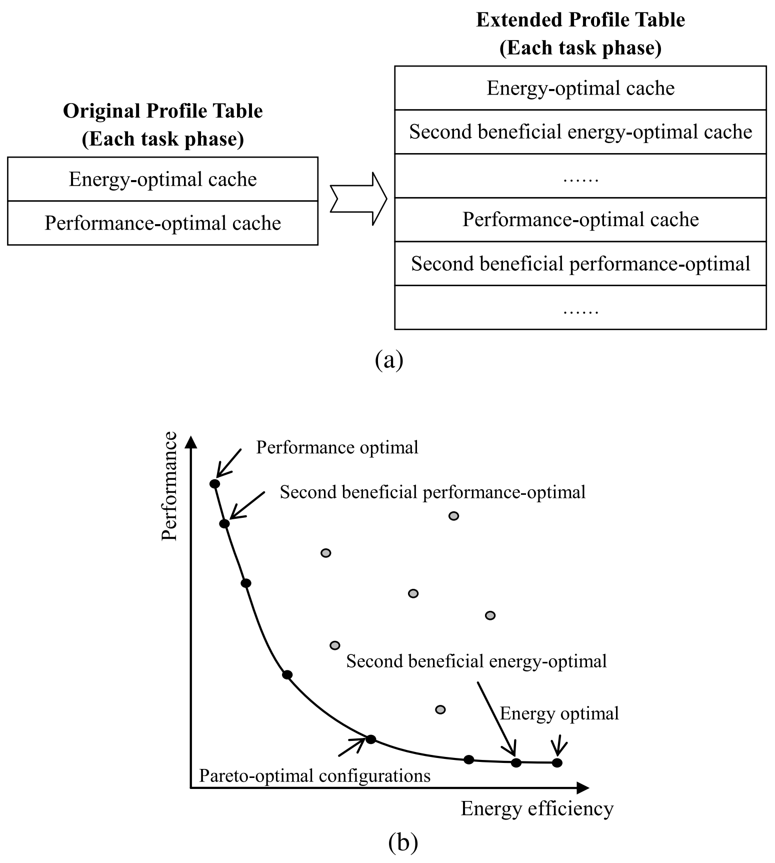

Extended Profile Table
(Each task phase)
Energy-optimal cache
Second beneficial energy-optimal cache
……
Performance-optimal cache
Second beneficial performance-optimal
……
Original Profile Table
(Each task phase)
Energy-optimal cache
Performance-optimal cache
(a)
Performance
Performance optimal
Second beneficial performance-optimal
Second beneficial energy-optimal
Energy optimal
Pareto-optimal configurations
Energy efficiency
(b)

Fig. 8. (a) Storing multiple optimal cache configurations for each task phase; (b) second beneficial optimal cache selection on the Pareto-optimal curve.

phase in the profile table, the scheduler would be capable of selecting the lowest energy cache configuration that is capable of meeting time constraints of all the existing tasks in the system. Thus, this is a trade-off between potential energy savings and system overhead in the form of table storage and scheduler complexity. Note that storing information for one more cache configuration in the table would potentially double the area overhead as well as increase power consumption and access time. Section 4.4 provides experimental results for this approach.

4. EXPERIMENTS

4.1 Experiments Setup

To quantify energy savings using our approaches, we examined selected benchmarks from the MediaBench [Lee et al. 1997] (mostly multimedia applications) and the EEMBC Automotive [EEMBC 2000] benchmark suites, representing typical tasks that might be present in a soft real-time system. All applications were executed with the default input sets provided with the benchmarks suites.

We utilized the configurable cache architecture for the L1 cache developed by Zhang et al. [2005] with a four-bank cache with base size of 4KB, which offers sizes of 1KB, 2KB, and 4KB, line sizes ranging from 16 bytes to 64 bytes, and associativity of one-way, two-way, and four-way. For comparison purposes, we define the *base cache* configuration to be a 4KB, two-way set associative cache with a 32-byte line size—a reasonably common configuration that meets the needs of the benchmarks studied. The L2 cache is set to a 64KB unified cache with four-way associativity and a 32-byte line size. Our energy model, adopted from the one used in Zhang et al. [2005], calculates both dynamic and static energy consumption, memory latency, processor stall energy, and main memory fetch energy. Let E_{dyn} and E_{sta} denote the dynamic energy and static energy of the cache subsystem, respectively. The total cache energy consumption is $E_{cache} = E_{dyn} + E_{sta}$. Specifically, we have

$$E_{dyn} = num\_accesses \cdot E_{access} + num\_misses \cdot E_{miss}; \tag{2}$$

$$E_{miss} = E_{offchip\_access} + E_{\mu P\_stall} + E_{block\_fill}; \tag{3}$$

$$E_{sta} = P_{sta} \cdot CC \cdot t_{cycle}, \tag{4}$$

where E_{access}, E_{miss}, and P_{static} are the energy required per cache access, per cache miss, and static power consumption, respectively. E_{access} and P_{static} are collected from CACTI 4.2 [HP 2008] with a 0.18 μm technology for all cache configurations. Following Zhang et al. [2005], we represent E_{miss} as the sum of the energy consumptions for fetching data from off-chip memory $E_{offchip\_access}$, processor stall energy due to a cache miss $E_{\mu P\_stall}$, and cache line refilling after a miss E_{block_fill}. CC denotes the number of clock cycles that is required to execute the task, and t_{cycle} is the length of each clock cycle. The access latency (e.g., in nanoseconds) to read data from the cache remains the same when we reconfigure the cache, because the clock frequency is determined by the base cache size. The data transfer time during a cache miss is determined by the cache line size as well as the bandwidth between the memory levels. In general, larger line sizes will lead to more data transfer cycles, thus higher access latencies. This variance for both L1 and L2 caches is incorporated in our model considering different miss cycles for cache configurations with various line sizes. We adopt these values from the study in Zhang et al. [2005]. To obtain cache hit and miss statistics, we used the SimpleScalar toolset [Burger et al. 1996] to simulate the applications. We assume an in-order issue core with a four-stage pipeline, which supports out-of-order completion, but the pipeline is stalled whenever a data hazard is detected. The simulator also supports speculation and a branch predictor with a two-bit saturating counter. We use the PISA architecture in our experiments, and the compiler is the default little-endian PISA compiler (sslittle-na-sstrix-gcc) provided with SimpleScalar 3.0 with cc options CFLAGS= −O −I$(srcdir). To populate the static profile tables for each task, we utilize SimpleScalar's external I/O trace files (eio files), checkpointing, and fastforwarding capabilities. This method allows for every benchmark phase to be individually profiled via fastforwarding execution to each potential preemption point. In our experiments, we used partition factors ranging from four to seven. Driven by Perl scripts, the design space of 18 cache configurations is exhaustively explored during static analysis to determine the energy-, performance-, and deadline-aware energy-optimal cache configurations for each phase of each benchmark.

4.2 Results

To model sample real-time embedded systems with multiple executing tasks, we created seven different task sets, as shown in Table IV. In each task set, the three selected benchmarks have comparable dynamic instruction sizes in order to avoid behavioral domination by one relatively large task. For system simulation, task arrival times and deadlines are randomly generated. To achieve an effective and fair comparison, we make the system utilization ratio close to the schedulability condition [Liu 2000]. We examine varying preempting points and average these values so that our results represent a generic degree of scheduling decisions.

Table IV. Benchmark Task Sets

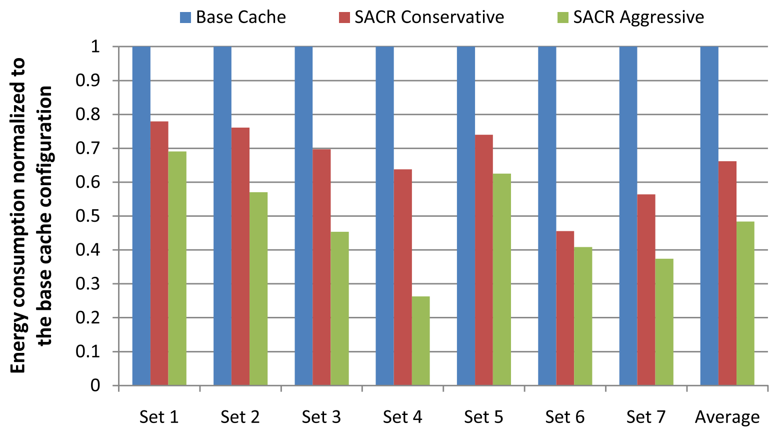
	Task 1	**Task 2**	**Task 3**
Task Set 1	epic*	pegwit*	rawcaudio*
Task Set 2	cjpeg*	toast*	mpeg2*
Task Set 3	A2TIME01**	AIFFTR01**	AIFIRF01**
Task Set 4	BITMNP01**	IDCTRN01**	RSPEED01**
Task Set 5	djpeg*	rawdaudio*	untoast*
Task Set 6	BaseFP01**	CACHEB01**	IIRFLT01**
Task Set 7	TBLOOK01**	TTSPRK01**	PUWMOD01**

Note: **MediaBench; **EEBMC.*

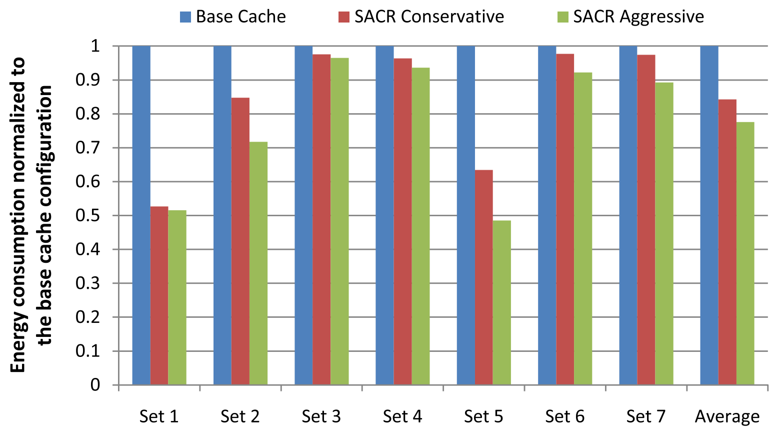
We compare the energy consumption for each task set using different schemes: a fixed base cache configuration, the conservative approach, and the aggressive approach. Energy consumption is normalized to the fixed base cache configuration such that a value of 1 represents our baseline. Figure 9 presents energy savings for the instruction and data cache subsystems. Energy savings in the instruction cache subsystem ranges from 22% to 54% for the conservative approach, while it reaches as high as 74% for the aggressive approach. Energy savings average 33% and 52% for the conservative and aggressive approaches, respectively. In the data cache subsystem, energy savings are generally less than that of the instruction cache subsystem due to less variation in cache configuration requirements. In the data cache subsystem, energy savings range from 15% to 47% for the conservative approach, while it reaches as high as 64% for the aggressive approach, and the averages are 16% and 22% for the conservative and aggressive approaches, respectively.

It is worth investigating the insights behind the experimental results: why instruction caches and data caches reveal such different energy savings when executing tasks from different benchmark suites (MediaBench and EEMBC). Note that we use benchmarks from MediaBench in task sets 1 and 2 and benchmarks from EEMBC in task sets 3 and 4. As shown in Figure 9, task set 1, for example, has more energy savings in the data cache than in the instruction cache using the aggressive approach. By evaluating the properties of each benchmark in that task set, we found that these benchmarks have common characteristics in their energy-optimal and performance-optimal cache configurations stored in the profile table. To illustrate this, we sort the tasks' cache configurations by their energy consumption as well as performance. Figure 10 depicts the layout of each configuration, ranking by its energy and performance for benchmark epic.[4] We can see that in the data cache, the chosen energy-optimal cache's performance and performance-optimal cache's energy consumption are relatively much better than the instruction cache. The higher the performance an energy-optimal cache configuration has, the higher the chance that it will be chosen by the scheduler. On the other hand, the less energy a performance-optimal cache configuration consumes, the less penalty (extra energy consumption) it has to pay when the scheduler has to choose the performance-optimal configuration due to tight timing constraints. These two factors explain why for test case 1, the data cache reveals more energy savings than the instruction cache. For those task sets containing benchmarks from EEBMC, the situation is the opposite. Task sets 3 and 4 do very well in the instruction cache but show very little energy savings in the data cache. Figure 11 illustrates the reason for this observation. Again, though only A2TIME01 is shown, we found almost all the

[4]Due to space limits, results for one task in test set 1 are shown here; however, other tasks in that set also have a similar pattern. For the same reason, even though only the results for the entire benchmark are shown here, other phases also show the same property.



(a) Instruction cache.

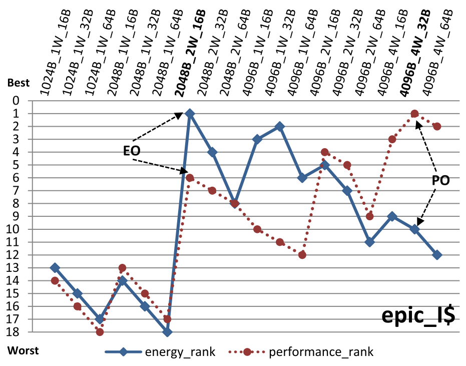


(b) Data cache.

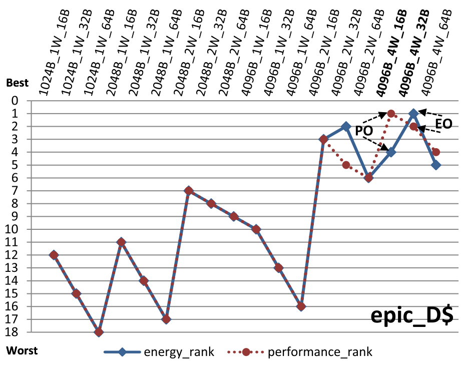
Fig. 9. Cache subsystem energy consumption normalized to the base cache configuration for each task set.

benchmarks in EEMBC have the same property. In the instruction cache, the performance of the energy-optimal cache is very close to that of the performance-optimal cache. Similarly, the energy consumption of the performance-optimal cache is very close to that of the energy-optimal cache. Interestingly, in many cases they are the same cache configuration, for example, A2TIME01 in Figure 11(a). However, in the data cache, the energy-efficient caches and performance-efficient caches differ tremendously. For this reason, benchmarks from EEMBC show poor results for the data cache.

It is also helpful to discuss how the cache miss rate plays its role in the cache model and thus affects the optimal cache configuration variations. Figure 12 shows the miss rates for epic, which explains the insights behind Figure 10, showing each data cache configuration behaving similarly in terms of both performance and energy efficiency (e.g., Figure 10(b)), while the instruction cache behaves just the opposite. The reason for $4KB\_4W\_16B$ and $4KB\_4W\_32B$ being superior in both energy and performance is the following. On one hand, the benchmark's data region in the footprint is relatively large, and thus the capacity of the data cache is critical. In other words, configurations with smaller sizes cannot satisfy the benchmark's footprint and thus suffer from

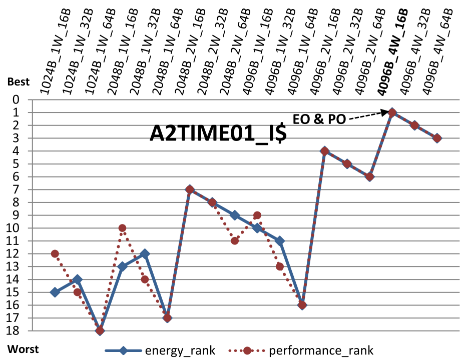


(a) Instruction cache (epic).

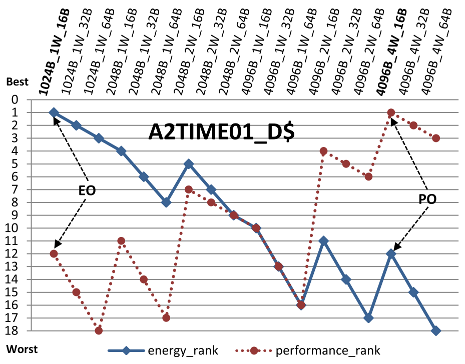


(b) Data cache (epic).

Fig. 10. Cache configuration candidate's energy and performance rank layout.



(a) Instruction cache (A2TIME01).



(b) Data cache (A2TIME01).

Fig. 11. Cache configuration candidate's energy and performance rank layout.

high miss rates. Therefore, with the same associativity, configurations with a larger capacity always win over those with smaller size in both performance and energy. On the other hand, as shown in Figure 12, 1KB configurations with two-way associativity[5] have similar miss rates as 2KB direct-mapped caches, while 2KB and two-way associativity configurations have lower miss rates than 4KB direct-mapped caches. Therefore, the temporal locality of the benchmark is reflected in the number of conflict misses (which is further reflected in the desired cache associativity) and also plays an important role in determining optimal cache configurations. The code region in the footprint is relatively small and thus can be easily satisfied; each configuration will show similar low miss rates, thus smaller configurations could win in energy efficiency due to their low power dissipation.

To illustrate the effect of leakage power on the optimal cache configuration, Figures 13(a) and 13(b) show both dynamic and static energy consumption for various cache configurations for dijkstra and A2TIME01, respectively. For dijkstra, the smallest cache configuration overall wins, since larger-sized configurations do not show much efficiency in dynamic energy while resulting in larger static energy. However, for A2TIME01, larger cache configurations outperform smaller configurations due to

[5]Although 1KB cache with two-way associativity is not valid in our reconfigurable cache architecture, we include it here for illustration purposes only.

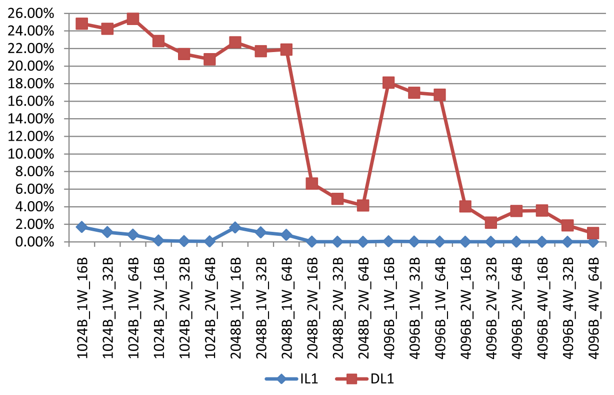


Fig. 12. Miss rate for epic.

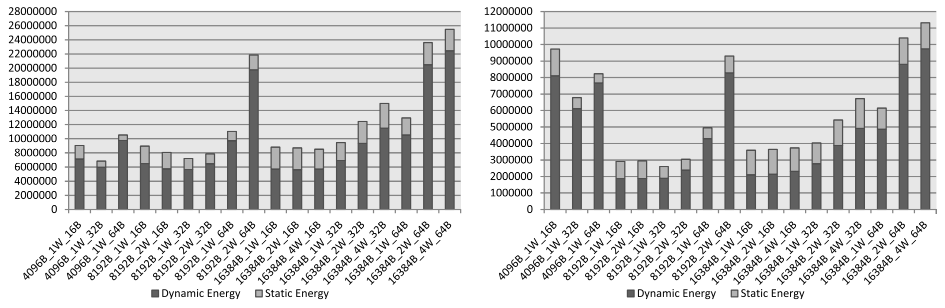

(a) dijkstra. (b) A2TIME01.

Fig. 13. Cache energy consumption decomposition.

significant reductions in dynamic energy consumption. This also explains why different benchmarks favor different cache configurations.

4.3 Suitability of Statically Determined Configurations

A system's performance variations when using our approaches are shown in Tables V and VI. We keep track of each task's performance during execution and determine the percentage of the task's jobs whose performance are higher (and lower but deadlines are met) using the selected cache configuration as compared to the base cache configuration. As discussed in Section 3.4.1, the cache configuration selected by our approach (nearest-neighbor nature) may possibly be inefficient in performance for the execution period between the actual preemption points. The percentage of deadline misses are also provided for each task to evaluate the system service level. Though lower performance jobs potentially have an impact on the system performance, they are not harmful, since no task deadline is missed. As the results show, our approach achieves significant energy savings at the expense of a small amount of task deadline misses which are acceptable in soft real-time systems. For example, among epic's all executions (jobs), 75% of them took a shorter period of time using the cache configurations selected by the conservative approach as compared to using the base cache configuration, while 21% of them took a longer time but still met the time constraints. Only 4% of all its jobs actually miss their deadlines. As Table V demonstrates, our conservative approach leads to very minor deadline misses (0–4%). Our aggressive approach can

Table V. Task Performance Variations for the Conservative Approach

Task Sets	Tasks	Higher performance jobs	Lower performance jobs	Deadline misses
1	epic	75%	21%	4%
	pegwit	99%	1%	0%
	rawcaudio	94%	3%	3%
2	cjpeg	94%	5%	1%
	toast	89%	4%	7%
	mpeg2	94%	2%	4%
3	A2TIME01	98%	2%	0%
	AIFFTR01	82%	15%	3%
	AIFIRF01	99%	1%	0%
4	BITMNP01	100%	0%	0%
	IDCTRN01	96%	2%	2%
	RSPEED01	99%	1%	0%

Table VI. Task Performance Variations for the Aggressive Approach

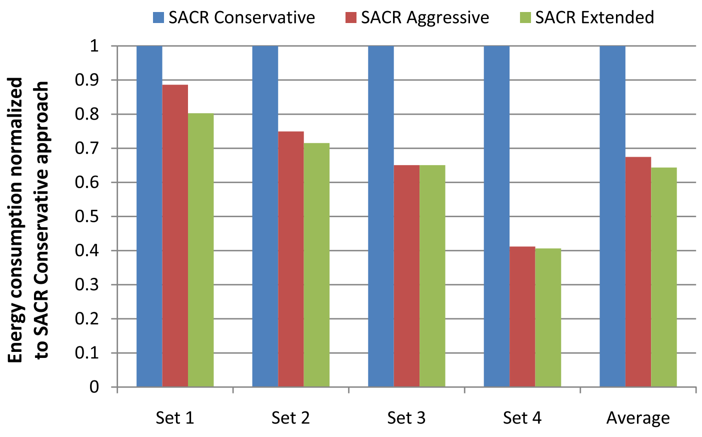
Task Sets	Tasks	Higher performance jobs	Lower performance jobs	Deadline misses
1	epic	63%	29%	8%
	pegwit	89%	10%	1%
	rawcaudio	76%	12%	12%
2	cjpeg	90%	6%	4%
	toast	72%	16%	12%
	mpeg2	75%	17%	8%
3	A2TIME01	94%	2%	3%
	AIFFTR01	52%	30%	18%
	AIFIRF01	97%	2%	1%
4	BITMNP01	62%	27%	11%
	IDCTRN01	94%	3%	3%
	RSPEED01	91%	2%	7%

Table VII. Current Phases of Deadline Violated Tasks When the Task is Discarded

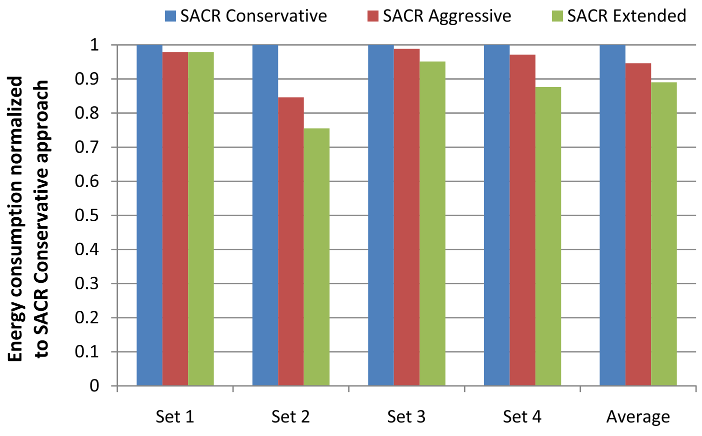
Task Sets	Tasks	CP = 1	CP = 2	CP = 3
1	epic	23%	54%	23%
	pegwit	0%	0%	100%
	rawcaudio	33%	34%	33%
2	cjpeg	14%	34%	52%
	toast	10%	25%	65%
	mpeg2	18%	32%	50%

generate a drastic reduction in energy requirements with a slightly increased number of missed deadlines (1%–18%).

To illustrate how early/late in the execution the deadlines are missed, for each low-priority job that is discarded, we collected its current phase (CP), as shown in Table VII. In other words, among all the jobs that missed their deadlines (e.g., 4% of



(a) Instruction cache.



(b) Data cache.

Fig. 14. The effect of using the extended profile table: cache subsystem energy consumption normalized to the conservative approach for each task set.

all jobs), different jobs are dropped at different stages of execution (CP). For example, in the case of epic, among the 8% of jobs that are dropped, 23% of them have executed over one-fourth (CP = 1), 54% of them have executed over half (CP = 2), and 23% of them are over three-fourths (CP = 3).

4.4 Impact of Storing Multiple Cache Configurations

As discussed in Section 3.5, storing multiple beneficial cache configurations may lead to more energy savings. We explore the effect of using an extended profile table by running task set 1–4 in Table IV. The profile table size is doubled to accommodate the second beneficial energy- and performance-optimal cache configurations. Algorithm 2 is modified to be aware of this extension. We call this method the *Extended* approach, and Figure 14 shows its energy consumption compared to the conservative and aggressive approaches. On average, the extended approach achieves 4.6% more energy savings than the aggressive approach in the instruction caches while 5.9% more in the data caches. In some cases (like task set 3 in the instruction cache and task set 1 in the data cache), no extra energy savings is observed due to the lack of beneficial cache configurations.

As already discussed in Section 3.5, the extended profile table will cause an exponential increase in system overheads. The energy overhead of the profile table can be safely ignored, because it only accounts for a small proportion of the gained energy savings. However, the increase in the area and access time of the table affects the feasibility of applying the extended approach. When the number of different tasks is relatively small such that the system overhead is not large, the extended approach is favored over the other two approaches. Since it is common to have a large number of tasks in the system, applying the extended approach may not be a good idea in these cases, because the profile table's area could exceed the chip area constraints, and increased access time may impact the system's critical path. In extreme cases, it may lead to longer clock cycle length and lower system clock frequency, which should obviously be avoided.

4.5 Analysis of Input Variations

A program's cache behavior, especially in the data cache, can vary when using different inputs. Essentially, inputs can vary in size, structure, and contents. For example, different inputs may drastically affect the program's dynamic execution path (such as the number of loop iterations), thus both energy- and performance-optimal caches may differ from what are stored in the profile table.

Obviously, it is impossible to exhaustively explore all potential inputs. Energy-aware task scheduling techniques face the same problem. In real-time systems, as discussed in Section 2.1, the scheduler should be supplied with the task set information, which includes the task's execution time (in cycles). The potential solutions include use of i) a fixed input set (execution time is known beforehand) [Hu and Marculescu 2004; Rong and Pedram 2008], ii) worst-case execution time (WCET) [Jejurikar et al. 2004; Seo et al. 2004; Shin et al. 2001; Zhang et al. 2007] and iii) a probabilistic execution time distribution [Hong et al. 2006; Oh et al. 2008; Zhong and Xu 2005].

It is worth exploring how varying inputs would impact each; task's cache behavior. In our experiments, we examine inputs with different sizes and observe the variation in the optimal cache configurations. For the instruction cache, the energy-optimal cache configuration parameters (cache size, line size, and set associativity) reduce as the input size decreases. Results are similar for the data cache. The performance-optimal instruction cache configuration's line size reduces as input size decreases, but cache size and associativity remain the same. However, in this case, the data cache shows nondeterministic behavior. The reason for the variation in the instruction cache is the size of the critical data processing code sections, which accounts for 90% of the execution time (e.g., loops, etc.) and may be a comparatively small segment of the entire program due to the 90/10 rule. Since critical data processing code sections (the instruction cache working set) remains in the cache, the line size tends to be smaller in order to reduce the time spent on cache misses and, thus, static energy consumption. For the data cache, as the input size increases, spatial locality is more critical than temporal locality, thus the cache size nearly remains the same (or even decreases), but line size increases. It is important to note that drastic changes in input size are not usual in real-time systems. We also studied the impact of changing input patterns on our approach. We observed that a change in input pattern (data structure and the absolute values change but not the size) shows a minor impact on the cache behavior. Both energy- and performance-optimal cache configurations show very little variation.

The following experimental results support these arguments. We examined cjpeg from MediaBench. In the first set of experiments, we selected six different-sized input

Table VIII. Input Variation Exploration

a.ppm: Size of input: 8431 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
4096B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
4096B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
b.ppm: Size of input: 101484 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_16B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
c.ppm: Size of input: 306915 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
d.ppm: Size of input: 530895 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	4096B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
e.ppm: Size of input: 1476015 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_16B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	4096B_2W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_2W_64B	2048B_2W_32B	4096B_4W_16B
f.ppm: Size of input: 3832336 bytes			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_2W_64B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	2048B_2W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	2048B_2W_64B	2048B_2W_32B	4096B_4W_16B

image files (a.ppm, b.ppm, c.ppm, d.ppm, e.ppm, and f.ppm) and found the energy-performance-optimal cache configurations for both instruction and data caches with a partition factor of four, as shown in Table VIII. In the second experiment, we selected two similarly sized image files (man.ppm and woman.ppm) with different content and explored the cache behavior using partition factors of four, five, and six. As shown in Table IX, there is very little variation in terms of the optimal cache configuration selection for the two inputs. Therefore, our approach is applicable when the input for each task is known during design time so that it can be statically profiled. Our approach is also applicable when there are changes in input pattern. This is a realistic assumption for real-time systems.

4.6 Hardware Overhead

This section describes the overhead of implementing the profile table in hardware. The profile table is stored in SRAM and accessed by the cache tuner to fetch the cache configuration information. The size of the table depends on the number of tasks in the system and the partition factor used. For the conservative approach, each table entry

Table IX. Input Pattern Changes

man.ppm: Size of input: 336165 bytes			
Partition factor p = 4			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 5			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 6			
EO_icache	PO_icache	EO_dcache	PO_dcache
4096B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
woman.ppm: Size of input: 312999 bytes			
Partition factor p = 4			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 5			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B
Partition factor p = 6			
EO_icache	PO_icache	EO_dcache	PO_dcache
2048B_2W_16B	4096B_4W_32B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	4096B_4W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_32B	4096B_2W_16B	4096B_4W_16B
2048B_2W_16B	4096B_4W_64B	2048B_2W_32B	4096B_4W_16B
2048B_2W_16B	4096B_4W_16B	2048B_2W_32B	4096B_4W_16B

consists of five bits, since the configurable cache architecture used in this study offers 18 possible cache configurations. We have implemented the profile table using Verilog HDL and synthesized the table using Synopsis Design Compiler with a TSMC 0.18 cell library. We estimate a table lookup frequency of once per three million nanoseconds during dynamic power computation, which means that there is a table lookup every one million instructions using a 500 MHz CPU with an average CPI of 1.5. It is clearly an overestimation (which is safe), since the benchmarks we used have around 10 to

Table X. Overhead of Different Lookup Tables (180nm Technology)

Table size (# of entries)	**Area ($\mu \mathbf{m}^2$)**	**Dynamic Power (μW)**	**Leakage Power (μW)**	**Critical Path Length**
64	61,416	38.13	114.37	0.91
128	121,200	84.25	224.90	0.91
256	244,520	187.68	461.30	1.08
512	483,416	327.90	904.70	1.20

Table XI. Overhead of Different Lookup Tables (65nm Technology)

Table size (# of entries)	**Area ($\mu \mathbf{m}^2$)**	**Dynamic Power (μW)**	**Leakage Power (μW)**
64	6,756	12.23	154.52
128	13,332	27.02	303.86
256	26,897	60.19	623.25
512	53,176	105.16	1,222.32

200 million dynamic instructions. Table X illustrates our results. Each row in the table indicates the area, dynamic power, leakage power, and critical path length for the profile table with different sizes. We also calculated the overhead using 65nm technology, as shown in Table XI. We observed that on average for each task set, the energy overhead of our approach only accounts for less than 0.02% (450 nJ compared to 2,825,563 nJ) of the total energy savings. The aggressive approach requires more bits per lookup table entry (74 bits[6]). However, Tables X and XI illustrate that the power dissipation is approximately linearly proportional to the table size. Therefore, even if the table entry size is increased by 15 times (5 bits to 74 bits), the total energy overhead is no more than 0.3% of the total energy savings. Therefore, we can safely conclude that the overhead for the profile tables are negligible compared to the energy savings for both the conservative and aggressive approaches.

5. CONCLUSIONS

Dynamic reconfiguration techniques are widely used in designing efficient embedded systems. Dynamic cache reconfiguration is a promising approach for improving both energy consumption and overall performance. The contribution of this article is a novel scheduling-aware dynamic cache reconfiguration technique for soft real-time systems. To the best of our knowledge, this is the first approach integrating dynamic cache reconfiguration into real-time systems. Our methodology employs an ideal combination of static analysis and dynamic tuning of cache parameters with minor or no impact on timing constraints. Our experiments demonstrated a 50% reduction on average in the overall energy consumption of the cache subsystem in soft real-time embedded systems. Our future work includes application of our approach to hard real-time systems.

[6]74 bits are needed to store both energy- and performance-optimal cache configurations (5 + 5 bits) as well as the corresponding execution times (32 + 32 bits).

REFERENCES

ANDERSSON, B., BLETSAS, K., AND BARUAH, S. 2008. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Proceedings of the Real-Time Systems Symposium*. 385–394.

BENINI, L., BOGLIOLO, R., AND MICHELI, G. D. 2000. A survey of design techniques for system-level dynamic power management. *IEEE Trans. VLSI Syst. 8*, 299–316.

BURGER, D., AUSTIN, T. M., AND BENNETT, S. 1996. Evaluating future microprocessors: The simplescalar tool set. Tech. rep., University of Wisconsin-Madison, Madison, WI.

BUTTAZZO, G. 1995. *Hard Real-Time Computing Systems*. Kluwer, Berlin, Heidelberg.

EEMBC. 2000. EEMBC, The Embedded Microprocessor Benchmark Consortium. http://www.eembc.org.

GORDON-ROSS, A. AND VAHID, F. 2004. Automatic tuning of two-level caches to embedded applications. In *Proceedings of the Design, Automation and Test in Europe Conference*. 208–213.

GORDON-ROSS, A., VAHID, F., AND DUTT, N. 2005. Fast configurable-cache tuning with a unified second-level cache. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'05)*. 323–326.

GORDON-ROSS, A., VIANA, P., VAHID, F., NAJJAR, W., AND BARROS, E. 2007. A one-shot configurable-cache tuner for improved energy and performance. In *Proceedings of the Design, Automation and Test in Europe Conference*. 755–760.

HENNESSY, J. AND PATTERSON, D. 2003. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Waltham, MA.

HONG, I., KIROVSKI, D., QU, G., POTKONJAK, M., AND SRIVASTAVA, M. B. 1999. Power optimization of variable-voltage core-based systems. *IEEE Trans. Comput.-Aided Des. Integr. Cir. Syst. 18*, 1702–1714.

HONG, S., YOO, S., JIN, H., CHOI, K., KONG, J., AND EO, S. 2006. Runtime distribution-aware dynamic voltage scaling. In *Proceedings of the International Conference on Computer-Aided Design*. 587–594.

HP. 2008. CACTI, HP Laboratories Palo Alto, CACTI 5.3. http://www.hpl.hp.com/.

HU, J. AND MARCULESCU, R. 2004. Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In *Proceedings of the Design, Automation and Test in Europe Conference*. 234–239.

JEJURIKAR, R. AND GUPTA, R. 2005. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the Design Automation Conference*. 111–116.

JEJURIKAR, R. AND GUPTA, R. 2006. Energy-aware task scheduling with task synchronization for embedded real-time systems. *IEEE Trans. Comput.-Aided Des. Integr. Cir. Syst. 25*, 1024–1037.

JEJURIKAR, R., PEREIRA, C., AND GUPTA, R. K. 2004. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the Design Automation Conference*. 275–280.

KIM, H., SOMANI, A. K., AND TYAGI, A. 2000. A reconfigurable multi-function computing cache architecture. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. 85–94.

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W. H. 1997. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the International Symposium on Microarchitecture*. 330–335.

LEUNG, L., TSUI, C., AND HU, X. S. 2005. Exploiting dynamic workload variation in low energy preemptive task scheduling. In *Proceedings of the Design, Automation and Test in Europe Conference*. 634–639.

LIU, J. 2000. *Real-Time Systems*. Prentice Hall, Upper Saddle River, NJ.

MALIK, A., MOYER, B., AND CERMAK, D. 2000. A low power unified cache architecture providing power and performance flexibility. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 241–243.

MODARRESSI, M., HESSABI, S., AND GOUDARZI, M. 2006. A reconfigurable cache architecture for object-oriented embedded systems. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*. 959–962.

NACUL, A. C. AND GIVARGIS, T. 2004. Dynamic voltage and cache reconfiguration for low power. In *Proceedings of the Design, Automation and Test in Europe Conference*. 21376.

OH, S., KIM, J., KIM, S., AND KYUNG, C. 2008. Task partitioning algorithm for intra-task dynamic voltage scaling. In *Proceedings of the International Symposium on Circuits and Systems*. 1228–1231.

PUANT, I. 2002. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*.

PUANT, I. AND DECOTIGNY, D. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*. 114–125.

PUANT, I. AND PAIS, C. 2007. Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison. In *Proceedings of the Design, Automation and Test in Europe Conference*. 1484–1489.

QUAN, G. AND HU, X. S. 2007. Energy efficient dvs schedule for fixed-priority real-time systems. *ACM Trans. Des. Autom. Electron. Syst. 6*, 1–30.

RONG, P. AND PEDRAM, M. 2008. Energy-aware task scheduling and dynamic voltage scaling in a real-time system. *J. Low Power Electron. 4*, 1–10.

SEGARS, S. 2001. Low power design techniques for microprocessors. In *Proceedings of the International Solid State Circuit Conference*.

SEO, J., KIM, T., AND CHUNG, K. 2004. Profile-based optimal intra-task voltage scheduling for hard real-time applications. In *Proceedings of the Design Automation Conference*. 87–92.

SETTLE, A., CONNORS, D., AND GIBERT, E. 2006. A dynamically reconfigurable cache for multithreaded processors. *J. Embed. Comput. 2*, 221–233.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., SAIR, S., AND CALDER, B. 2003. Discovering and exploiting program phases. In *Proceedings of the International Symposium on Microarchitecture*. 84–93.

SHIN, D., KIM, J., AND LEE, S. 2001. Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the Design Automation Conference*. 438–443.

SPRUNT, B. 1990. Aperiodic task scheduling for real-time systems. Ph.D. dissertation, Carnegie Mellon University, Pittsburg, PA.

STASCHULAT, J., SCHLIECKER, S., AND ERNST, R. 2005. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the Euromicro Conference on Real-Time Systems*. 41–48.

TAN, Y. AND MOONEY, V. J. 2007. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embed. Comput. Syst. 6,* 7.

VARMA, A., DEBES, E., KOZINTSEV, I., AND JACOB, B. 2005. Instruction-level power dissipation in the intel xscale embedded microprocessor. In *Proceedings of the SPIE 17th Annual Symposium on Electronic Imaging Science & Technology*.

WANG, W. AND MISHRA, P. 2009. Dynamic reconfiguration of two-level caches in soft real-time embedded systems. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. 145–150.

WOLFE, A. 1993. Software-based cache partitioning for real-time applications. In *Proceedings of the International Workshop on Responsive Computer Systems*.

ZHANG, C., VAHID, F., AND LYSECKY, R. 2004. A self-tuning cache architecture for embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference*.

ZHANG, C., VAHID, F., AND NAJJAR, W. 2005. A highly configurable cache for low energy embedded systems. *ACM Trans. Embed. Comput. Syst. 6*, 362–387.

ZHANG, S., CHATHA, K., AND KONJEVOD, G. 2007. Approximation algorithms for power minimization of earliest deadline first and rate monotonic schedules. In *Proceedings of the International Symposium on Low Power Electronics and Design*. 225–230.

ZHONG, X. AND XU, C. 2005. Energy-aware modeling and scheduling of real-time tasks for dynamic voltage scaling. In *Proceedings of the International Real-Time Systems Symposium*. 366–375.

Received February 2009; revised October 2009, March 2010, October 2010; accepted March 2011