SACR: Scheduling-Aware Cache Reconfiguration for Real-Time Embedded Systems

Weixun Wang and Prabhat Mishra
Department of Computer and Information Science and Engineering
University of Florida, Gainesville, FL
wewang@cise.ufl.edu, prabhat@cise.ufl.edu

Ann Gordon-Ross
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL
ann@ece.ufl.edu

Abstract

Dynamic reconfiguration techniques are widely used for efficient 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 embedded systems since dynamic analysis may adversely affect tasks with real-time constraints. This paper presents a novel approach for implementing cache reconfiguration in soft real-time systems by efficiently leveraging static analysis during execution to both minimize energy and maximize performance. 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 (up to 74%) reduce the overall energy consumption of the cache hierarchy in soft real-time systems.

1. Introduction

Design and optimization of real-time embedded systems have been widely studied over the last few decades. These systems require unique design considerations due to time constraints placed on the tasks. Hard real-time system tasks have deadlines and tasks 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 [4][12]. 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, multimedia, and housekeeping devices. Tasks in these systems remain effective even if their deadlines are not guaranteed to be met. Minor deadline misses may result in temporary service or quality degradation, but will not lead to incorrect behavior.

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 [2]. Alternatively, Dynamic Voltage Scaling (DVS) [8] methods can be used to reduce the clock frequency such that the tasks execute slowly but do not violate their deadlines [10][16].

In recent years, 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, and/or balance overall system behavior. One such tunable component is the cache hierarchy. An efficient reconfigurable cache framework and tuning algorithms are proposed in [7].

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 candidates. Furthermore, any change in cache configuration on-the-fly may 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. 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 very minor impacts to user experiences. Our proposed research focuses on soft real-time systems.

To the best of our knowledge, this is the first approach in exploiting dynamic reconfigurable caches in real-time systems. This paper presents a novel methodology for using reconfigurable caches in real-time systems with preemptive tasks. Our proposed methodology, Scheduling-Aware Cache Reconfiguration (SACR), provides an efficient and near optimal cache tuning strategy based on static program profiling for both statically and dynamically scheduled real-time systems. The goal is to optimize energy consumption with performance considerations via reconfigurable cache tuning while ensuring that the majority of task deadlines are met.

The rest of the paper 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 the paper.

2. Related Work

There are no prior works in the area of dynamic cache reconfiguration in real-time systems. Our proposed research is the first attempt in this direction. This section surveys the background literature in the following three related domains.

2.1 Real-Time Scheduling Techniques

Based on task properties and associated systems, scheduling algorithms can be classified into various types [12]. Earliest Deadline First (EDF) scheduling [4] and Rate Monotonic (RM) scheduling [12] 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 an acceptance test 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. In reality, these three kinds of tasks may exist simultaneously. In this work, we use EDF as the scheduling algorithm for tasks with only soft real-time constraints. However, RM is also applicable with minor changes in our approach.

2.2 Caches in Real-Time Systems

Incorporating caches into real-time embedded systems faces certain difficulties due to the unpredictability imposed on the system. Scheduling algorithms have difficultly calculating WCET for tasks since data access time cannot be predetermined in the presence of caches. A great deal of research efforts are directed at employing caches in real-time systems either by proving schedulability through WCET analysis and/or avoiding hazardous compulsory miss uncertainty altogether. WCET analysis is a static, design time analysis of tasks in the presence of caches to predict cache impact on task execution times [14]. Cache locking [15] is a technique in which useful cache lines are "locked" in the cache when a task is preempted so that these blocks 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 [19] is a similar but more aggressive approach where 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 [18] features tight delay estimation so that prediction accuracy is higher than in traditional WCET analysis. This improved accuracy can in turn result in a durable task schedule. Our approach is applicable to real-time systems that employ caches.

2.3 Reconfigurable Cache Architectures

In power constrained embedded systems, nearly half of the overall power consumption is attributed to the cache subsystem [13]. Fortunately, since applications require vastly different cache requirements in terms of cache size, line size, and associativity, research shows that specializing the cache to an application's needs can reduce energy consumption by 62% on average [6].

There exists much work in dynamic cache reconfiguration [1] [7]. The reconfigurable cache architecture proposed by Zhang et al. [20] imposes no overhead to the critical path, thus cache access time does not increase. Furthermore, the cache tuner consists of small custom hardware or a lightweight process running on a co-processor, which can alter the cache configuration via hardware or software configuration registers. The underlying cache architecture consists of four separate banks, each of which acts as a separate way. Way concatenation, which logically concatenates ways together, enables configurable associativity. Way shutdown effectively shuts down ways to vary cache size. Configurable line size, or block size, is achieved by setting a unit-length base line size and then fetching subsequent lines if the line size increases.

Given a runtime reconfigurable cache, determining the best cache configuration is a difficult process. Dynamic and static analyses are two possible techniques. With dynamic analysis, cache configurations are evaluated in system during runtime to determine the best configuration. However, it is inappropriate for real-time systems as it either imposes unpredictable performance or significant energy overhead, both due to the exploration of suboptimal cache configurations. During static analysis, various cache alternatives are explored and the best cache configuration is selected for each application in its entirety (application-based tuning) [7] or for each phase of execution within an application (phase-based tuning) [17]. Regardless of the tuning method, the predetermined best cache configuration (based on design requirements) may be stored in a look-up table or encoded into specialized instructions. The static analysis approach is most appropriate for real-time systems due to its non-intrusive nature. However, 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.

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 (SACR), provides an efficient and near optimal strategy for cache tuning based on static program profiling for both statically and dynamically scheduled real-time 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 optimal 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 demonstrative example to show how reconfigurable caches benefit real-time embedded 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* [1] configuration $\text{Cache}_{\text{base}}$ throughout all task executions. In the presence of a reconfigurable cache, as shown in Figure 1, different optimal cache configurations are determined for every *phase* of each task. For ease of illustration, we divide each task into two phases. $Phase_1$ starts from the beginning to the end, and $phase_2$ starts from the half position of the dynamic instruction flow (midpoint) to the end. The terms $\text{Cache}_{\text{T1}}^{1}$, $\text{Cache}_{\text{T1}}^{2}$, $\text{Cache}_{\text{T2}}^{1}$, and $\text{Cache}_{\text{T2}}^{2}$ represent the optimal cache configurations for $phase_1$ and $phase_2$ of task *T1* and task *T2*, respectively. These

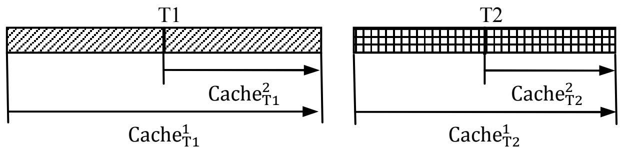
T1
T2
$\text{Cache}_{\text{T1}}^{2}$
$\text{Cache}_{\text{T1}}^{1}$
$\text{Cache}_{\text{T2}}^{2}$
$\text{Cache}_{\text{T2}}^{1}$

Figure 1: Cache configurations selected based on phases

[1] In this paper, we use the term "*base cache*" to refer to the cache used in typical real-time systems. Caches in such systems, as discussed in Section 2.2, are chosen to ensure durable task schedules.

configurations are chosen statically to be more energy efficient (with same or better performance), in their specific phases, than the global base cache, $\text{Cache}_{\text{base}}$.

Figure 2 illustrates how energy consumption can be reduced by using our approach in real-time systems. Figure 2(a) depicts a traditional system and Figure 2(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 $\text{Cache}_{\text{base}}$ exclusively. With a reconfigurable cache, the first part of *T1* executes using $\text{Cache}^{1}_{\text{T1}}$. Similarly, $\text{Cache}^{1}_{\text{T2}}$ is used for execution of *T2*. Note that the actual preemption point of *T1* is not exactly at the same place where we pre-computed the optimal cache configuration (midpoint). When *T1* resumes at time point *P2*, the cache is tuned to $\text{Cache}^{2}_{\text{T1}}$ since the actual preemption point is closer to the midpoint compared to the starting point.

The overall energy consumed using a reconfigurable cache results from the energy savings due to use of different energy optimal caches for each phase of task execution compared to using one global base cache in the traditional system. Our experimental results suggest that the proposed approach can reduce energy consumption up to 74% without introducing any performance penalty.

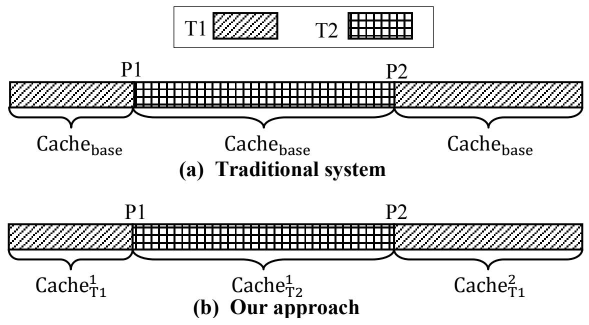

Figure 2: Dynamic cache reconfigurations for tasks T1 and T2.

3.2 Phase-based Optimal Cache Selection

This section describes our static analysis approach to determine 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 the entire application due to its own distinguishing behaviors as well as cold-start compulsory cache misses. Thus, the optimal cache configuration for the remainder of the task execution may be different. Figure 3 depicts the general case where a task is divided by *n* potential preemption points (0, P_1, P_2 … P_{n-1}). We define a static profiled *phase* as the period of time between a predefined *potential preemption point* (also called *partition points*) and task completion. Here, C_1, C_2 … C_n represent the optimal cache configuration (either energy or performance) for each phase, respectively. Again, the potential preemption points, which define phases, are decided during the static profiling stage and are not necessarily the same as actual preemption points observed during system execution.

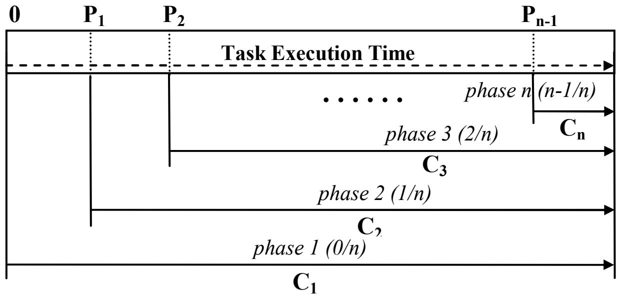

Figure 3: Task partitioning at *n* potential preemption points (P_i) resulting in *n* phases. Each phase comprises execution from the invocation/resumption point to task completion. C_i denotes the cache configuration used in each phase

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 partition points. The partition granularity is determined by dividing the total number of dynamically executed instructions by the partition factor. Smaller granularities result in finer grained configuration, and potentially greater energy savings. However, making granularity too fine would result in a prohibitively large look-up table which would not be feasible due to area constraints. Thus, a trade-off should be made to determine a reasonable partition factor based on energy-savings potential and acceptable overheads.

An important question is whether a larger partition factor (finer granularity) reveals more energy savings. Our experimental results show that once the partition factor is larger than a certain threshold for a task, more and more neighboring partitions share the same optimal cache configuration. This is evident due to the well-established 90/10 rule of 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 small loops. For each loop iteration, except the first and last, execution behavior is typically similar, thus resulting in the same optimal cache configuration for all iterations. For a loop with N iterations, the partition factor need only be large enough to capture all dynamic instructions of iterations 2 through N – 1, as any smaller granularity would capture a subset of iterations, each of which have the same optimal configuration. Clearly, if there is no variation, no energy savings is possible. Even if variations can be observed, according to our experiments, they only happen with very limited ranges, which means a minor energy saving is possible only when preemption/resumption takes place in these ranges (8% of the dynamic instruction flow on average). Thus, the goal of a system designer is to find a partition factor which leads to maximized energy reduction and minimizes the number of partition points that need to be stored. Based on our experience, a partition factor ranging from four to seven is sufficient to generate a static profile table that SACR can utilize efficiently.

The *profile table* is the output of static analysis that stores the potential preemption points and the corresponding optimal cache configurations for each task. Section 3.3 and 3.4 describe how this profile table is used during runtime of statically- as well as 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 (in hard real-time systems) for every execution phase 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 can tune the cache accordingly 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 feasible time constraint. In this section, we present two versions of our technique based on the nature of the target system.

3.4.1 SACR - Conservative Approach

In some soft real-time systems where time constraints are pressing, only an extremely small number of violations are tolerable. The SACR conservative approach could ensure that given a carefully chosen partition factor, almost every task could meet their deadlines with only 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 one with lowest energy consumption given the 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 deadline-aware energy-optimal cache configurations for each task phase. Table 1 (a) shows the profile table for task *i* with a partition factor *p*. $EO_i(n/p)$ represents the deadline-aware energy-optimal cache configuration for partition phase *n/p* (which means the execution of this phase is from the partition point of *n/p* until the end of the task) in task *i*. Here, *n/p* represents the *n*'th phase in the set of *p* phases. The total dynamic instruction count (*TIN*) refers to the number of dynamic instructions executed in a single run of that task.

During system execution, the scheduler maintains a task list keeping track of all existing tasks as shown in Table 1(b). In addition to the static profile table records from Table 1(a), runtime information such as arrival time (A_i), deadline (D_i), and remaining number of dynamic instructions (*RIN*) is recorded. This information is stored not only for the scheduler, but also for the cache tuner. When a newly arrived task begins execution, 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.

As indicated in Section 3.2.1, potential preemption points are pre-decided during the profile table generation process. However, it is highly unlikely that the actual preemptions will occur precisely on these potential preemption points. Hence, a *nearest-neighbor technique* 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 will be selected as the current cache configuration. As our experimental result shows, conservative SACR obtains significant energy savings with little or no impact on quality of service.

3.4.2 SACR - Aggressive Approach

In a soft real-time system with less pressing time constraints, a more aggressive version of SACR can reveal additional energy savings at the cost of possibly violating several low priority future task deadlines, while remaining 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 one with lowest energy consumption among all candidates) cache for every task phase. In order to assist dynamic scheduling, the profile table also includes the corresponding phase's approximate execution time (in cycles) for each configuration. Table 2(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.

Table 2(b) shows the task list entry for the aggressive

Table 1: (a) Static profile table and (b) Task list entry for task *i* for the conservative approach.

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

(a)

Task ID: i	**Partition Factor: p**
Arrival time (A_i)	**Deadline (D_i)**
Total Instruction Number (TIN)	**Remaining Instruction Number (RIN)**
$EO_i(0/p)$	
$EO_i(1/p)$	
$EO_i(2/p)$	
……	
$EO_i(p-1/p)$	

(b)

Table 2: (a) Static profile table and (b) task list entry for task *i* in the aggressive approach.

Task ID: i		**Partition Factor: p**	
Total Instruction Number (TIN)			
$EO_i(0/p)$	$EOT_i(0/p)$	$PO_i(0/p)$	$POT_i(0/p)$
$EO_i(1/p)$	$EOT_i(1/p)$	$PO_i(1/p)$	$POT_i(1/p)$
$EO_i(2/p)$	$EOT_i(2/p)$	$PO_i(2/p)$	$POT_i(2/p)$
……			
$EO_i(p-1/p)$	$EOT_i(p-1/p)$	$PO_i(p-1/p)$	$POT_i(p-1/p)$

(a)

Task ID: i		**Partition Factor: p**	
Arrival time (A_i)		**Deadline$_i$ (D_i)**	
Total Instruction Number (TIN)		**Remaining Instruction Number (RIN)**	
Current Phase (CP)			
$EO_i(0/p)$	$EOT_i(0/p)$	$PO_i(0/p)$	$POT_i(0/p)$
$EO_i(1/p)$	$EOT_i(1/p)$	$PO_i(1/p)$	$POT_i(1/p)$
$EO_i(2/p)$	$EOT_i(2/p)$	$PO_i(2/p)$	$POT_i(2/p)$
……			
$EO_i(p-1/p)$	$EOT_i(p-1/p)$	$PO_i(p-1/p)$	$POT_i(p-1/p)$

(b)

approach. The difference from the conservative approach (shown in Table 1(b)) is that every task list entry also holds a Current Phase (*CP_i*) identifier. CP_i denotes the partition point that this task's execution just passed and is useful for cache reconfiguration upon task resumption. In addition to the task list, the scheduler also maintains another runtime data structure called the ready task list (*RTL*), which contains an identifier representing each task currently ready to execute in the system.

To explain the SACR aggressive approach, we use an illustrative example in which there are three tasks, *T1*, *T2*, and *T3*, with 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 4 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 $P0 + EOT_{T1}(0/p) < D_{T1}$, 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:

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

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

If Equation 1 does not hold, the scheduler determines *T2*'s cache configuration C_{T2} as follows (assuming $POT_i(0/p) < D_i$ 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 to choose a cache configuration. This technique is similar in principle to the nearest neighbor approach 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*.

4. Experiments

4.1 Experimental Setup

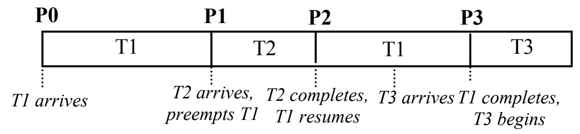

Figure 4: Task set and sample scheduling.

To quantify energy savings using SACR, we examined selected benchmarks from the MediaBench [11] and EEMBC Automotive [5] benchmark suites, representing typical tasks that might be present in a real-time system. All applications were executed with the default input sets provided with the benchmarks suites.

We utilized the configurable cache architecture developed by Zhang et al [20] with a four-bank cache of base size 4 KB, which offers sizes of 1 KB, 2 KB, and 4 KB, line sizes ranging from 16 bytes to 64 bytes, and associativity of 1-way, 2-way, and 4-way. For comparison purposes, we define the *base cache* configuration to be a 4 KB, 2-way set associative cache with a 32-byte line size, a reasonably common configuration that meets the needs of the benchmarks studied.

To obtain cache hit and miss statistics, we used the SimpleScalar toolset [3] to simulate the applications. Our energy model, adopted from the one used in [20], calculates both dynamic and static energy consumption, memory latency, CPU stall energy, and main memory fetch energy. We updated the dynamic energy consumption for each cache configuration using CACTI 4.2 [9].

To populate the static profile tables for each benchmark, we utilize SimpleScalar's external I/O trace files (eio file), 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 examined partition factors ranging from two to seven potential preemption points. 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 four different task sets as shown in Table 3. In each task set, the three selected benchmarks have comparable dynamic instruction counts in order to avoid behavioral domination by one relatively large task. For system simulation, task deadlines and priorities are as described in Section 3.4.2. We examine a varying set of preempting points and average these values so that our results represent a generic degree of scheduling decisions since task *T2* may preempt task *T1* at any point in time.

We compare the energy consumption for each task set using different schemes: a fixed base cache configuration, the SACR conservative approach, and the SACR aggressive approach. Energy consumption is normalized to the fixed base cache configuration such that value of 1 represents our baseline. Figure 5 and Figure 6 present energy savings for the instruction and data cache subsystems, respectively. Energy savings in the instruction cache subsystem ranges from 22% to 36% for the SACR conservative approach, while it reaches as high as 74% for the SACR aggressive approach. Energy savings average 28% and 51% for the SACR conservative and aggressive approaches, respectively. In the data cache subsystem, energy saving is generally less than that of the

Table 3: 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**

** MediaBench **EEMBC*

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 SACR conservative approach, while it reaches as high as 64% for the SACR aggressive approach, and average 17% and 22% for the SACR conservative and aggressive approaches, respectively.

The remainder of 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. The table entry 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 using Synopsis Design Compiler with TSMC 0.18 cell library. Table 4 illustrates our results. Each row in the table indicates the area, dynamic power, leakage power, and critical path length for profile table with different sizes. We assume a table lookup frequency of one million nanoseconds during dynamic power computation, which means there is a table lookup every five hundred thousand cycles (a reasonably frequency based on normal task sizes) using a 500MHz CPU. We observed that on average for each task set, the energy overhead of our approach only account for less than 0.02% (450 nJ comparing to 2825563 nJ) of the total energy savings. Therefore, the energy overhead of implementing the profile table is negligible compared to the energy savings produced by our approach.

Table 4: Overhead of different lookup tables

Table size (# of entries)	Area (μm^2)	Dynamic Power (nW)	Leakage Power (nW)	Critical Path Length (ns)
64	61416	134.40	114.37	0.91
128	121200	266.22	224.90	0.91
256	244520	544.73	461.30	1.08
512	483416	994.20	904.70	1.20

5. Conclusions

Dynamic reconfiguration techniques are widely used in designing efficient embedded systems. Dynamic cache reconfiguration is a promising approach to improve both energy consumption and overall performance. The contribution of this paper is a novel scheduling aware dynamic cache reconfiguration technique for soft real-time embedded systems. To the best of our knowledge, this is the first approach integrating dynamic cache reconfigurations into real-time embedded 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.

References

[1] D. H. Albonesi, “Selective cache ways: on demand cache resource allocation”, *Journal of Instruction Level Parallelism, May 2002.*

[2] L. Benini, G. De Micheli, “A survey of design techniques for system-level dynamic power management”, *TVLSI, 8(3):299-316, June 2000.*

[3] D. Burger, T. Austin, S. Bennet, “Evaluating future microprocessors: the simplescalar toolset”, *CS-TR-1308, University of Wisconsin, 2000.*

[4] G. Buttazzo, Hard Real-Time Computing Systems. *Kluwer 1995.*

[5] EEMBC, http://www.eembc.org.

[6] A. Gordon-Ross, F. Vahid, N. Dutt, “Automatic Tuning of Two-Level Caches to Embedded Applications”, *DATE,* page 10208, *2004.*

[7] A. Gordon-Ross, F. Vahid, N. Dutt, “Fast configurable-cache tuning with a unified second level cache, *ISLPED, pages 323-326, 2005.*

[8] I. Hong et al., “Power optimization of variable voltage core-based systems”, IEEE *TCAD, 18(12):1702-1714, December 1998.*

[9] HP Labs, CACTI 4.2, http://www.hpl.hp.com/

[10] R. Jejurikar, R. Gupta, “Energy-Aware Task Scheduling With Task Synchronization for Embedded Real-Time Systems”, IEEE *TCAD, 25(6):1024-103, June 2006.*

[11] C. Lee et al. “Mediabench: A tool for evaluating and synthesizing multimedia and communication systems”, *MICRO, 1997.*

[12] J. Liu, Real-Time Systems. Upper Saddle River, Prentice-Hall 2000.

[13] A. Malik et al., “A low power unified cache architecture providing power and performance flexibility”, *ISLPED, pages 241-243, 2000.*

[14] I. Puant, “Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems”, *ECRTS, 2002.*

[15] I. Puant et al., “Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems”, *RTSS, 114-125, 2002.*

[16] G. Quan, X. S. Hu, “Energy Efficient DVS Schedule for Fixed-Priority Real-Time Systems”, *ACM TECS,6(4),article 29, 2007.*

[17] T. Sherwood et al., “Discovering and exploiting program phases”, *IEEE Micro, December 2003.*

[18] Y. Tan, V. Mooney, “Timing Analysis for Preemptive Multitasking Real-Time Systems with Caches”, *ACM TECS, (6)1, article 7, 2007.*

[19] A. Wolfe, “Software-Based Cache Partitioning for Real-time Applications”, *IWRCS, 1993.*

[20] C. Zhang, F. Vahid, W. Najjar, “A Highly Configurable Cache for Low Energy Embedded Systems”, *ACM TECS, 6(4):362-387, 2005.*

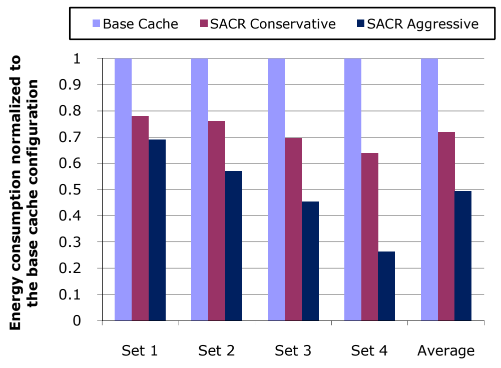

Figure 5: Instruction cache subsystem energy consumption normalized to the base cache configuration for each task set

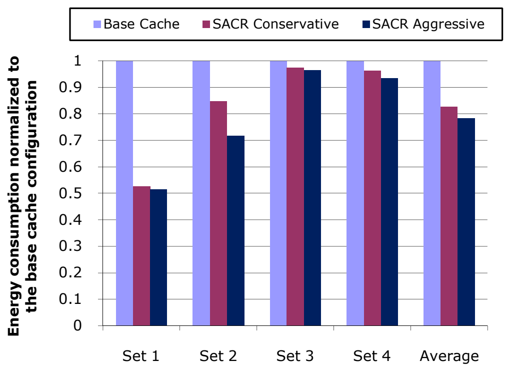

Figure 6: Data cache subsystem energy consumption normalized to the base cache configuration for each task set