Dynamic Reconfiguration of Two-Level Caches in Soft Real-Time Embedded Systems*

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

Abstract

Cache reconfiguration is a promising optimization technique for reducing memory hierarchy energy consumption with little or no impact on overall system performance. While cache reconfiguration is successful in desktop-based systems, it is not directly applicable in real-time systems due to timing constraints. Existing scheduling-aware cache reconfiguration techniques consider only one-level cache. It is a major challenge to dynamically tune multi-level caches since the exploration space is prohibitively large. This paper efficiently integrates cache reconfiguration in soft real-time systems with a unified two-level cache hierarchy. We utilize a set of exploration heuristics during our static analysis which effectively decreases the exploration time while keeps the generated profile results beneficial to be leveraged during runtime. Our experimental results have demonstrated 32 - 49% energy savings with minor impact on performance.

1 Introduction

Energy is one of the most stringent resources in embedded systems due to the fact that most of the devices are driven by batteries. Many low-power techniques and energy-aware algorithms target at different system components. Memory hierarchy is responsible for as much as 50% of the energy consumption of a microprocessor system [9]. According to Amdahl's law, such a large contribution to the overall energy consumption makes memory hierarchy a good candidate for optimization. Dynamic reconfiguration techniques offer the ability to meet each application's unique needs by tuning the parameters at runtime. Cache subsystem reconfiguration could lead to significant energy saving by meeting application's diverse cache requirements [1][17]. The working set of the application favors different cache sizes while its spatial locality determines favored line size. Furthermore, cache associativity reflects the application's temporal locality.

Real-time embedded systems have been widely studied over the last few decades with most of them focusing on scheduling, resource allocation and management. Real-time embedded systems have unique design considerations and optimization constraints that tasks must meet their deadlines. A task set is said to be schedulable if there exists a feasible schedule that can satisfy all timing constraints. Hence optimizations in real-time systems must be aware of the task *schedulability* in order to guarantee that the system's service quality does not get tampered. Hard real-time systems require that every task must be completed within its specified deadline and any violation will cause catastrophic consequences. Soft real-time systems, including multimedia systems, provide a more relaxed environment where a few tasks are allowed to be dropped or miss their deadlines. In other words, for soft real-time systems, minor deadline violation could result in temporary service degradation but the system remains effective. Earliest Deadline First (EDF) and Rate Monotonic (RM) [7] are the two most frequently referenced fundamental scheduling algorithms in real-time system research community.

It is a major challenge to apply cache reconfiguration in real-time systems. Both determining the appropriate cache configuration and tuning the cache hierarchy inevitably introduce runtime overhead if done dynamically. Changing cache configuration on-the-fly will also change the task's execution time, which may lead to unpredictable system behavior. Direct application of reconfigurable cache in real-time systems without careful consideration may not even be beneficial. Our previous work [15] have explored the use of one-level reconfigurable cache in soft real-time systems. However, it remains a challenge to dynamically tune multi-level caches since the exploration space is prohibitively large. This paper apply cache reconfiguration in soft real-time systems with a unified two-level cache hierarchy. We develop a set of exploration heuristics for our static analysis to effectively decrease the exploration time while keeping the generated profile results beneficial.

The rest of the paper is organized as follows. Section 2 presents a survey of related research areas. Section 3 describes background on configurable cache architecture and phase-based static profiling techniques. Section 4 describes our design space exploration and dynamic cache tuning technique. Section 5 presents our experimental results. Finally, Section 6 concludes the paper.

2 Related Work

There are many existing methods for general and application specific reconfigurable cache architectures. Motorola

*This work was partially supported by NSF CAREER Award 0746261.

M*CORE processor [9] provides way shut-down and way management, which has the ability to specify the content of each specific way (instruction, data, or unified way). Modarresssi et al. [10] developed a cache architecture which can be dynamically partitioned and resized to improve the performance of object-oriented embedded systems. Settle et al. [12] proposed a dynamically reconfigurable cache specifically designed for chip multi-processors. Zhang et al. [17] proposed an efficient and highly configurable cache architecture which imposes almost no overhead to the critical path. All of these approaches were studied in the context of desktop-based systems.

A lot of research efforts are spent in finding efficient automated techniques to reconfigure the cache hierarchy. Dynamic and static analysis are two possible ways to solve the problem. Both methods explore possible candidates and decide a profitable configuration to tune to at a given moment. If applications are unknown a priori, dynamic analysis is obviously the only option. However, its intrusive nature makes dynamic analysis infeasible in real-time systems since it imposes unpredictable performance overhead during exploration. In many cases, the applications are known during the design time. It makes static analysis attractive to real-time systems due to its non-intrusive nature. Static analysis explores design space to predetermine the best cache configuration for either the entire application or a part of it. The former strategy is called application-based tuning [4] while the latter is called phase-based tuning [13]. Previous works for tuning two-level cache hierarchy focused on design space reduction in desktop-based systems. Exploration heuristics introduced in [3] and [4] are designed for a configurable cache hierarchy with separate level-two caches [9] and with a unified level-two cache, respectively. However, none of these works is applicable to real-time systems.

While being ubiquitous in nearly all desktop level computing systems, incorporating caches into real-time embedded systems is still a hotspot issue. The difficulty mainly comes from the unpredictability nature of caches in terms of timing behavior. Fortunately, a great deal of research efforts have been carried out to employ caches in real-time systems. Puant et al. [11] present a technique in which cache lines in use are "locked" when a task is preempted so that these blocks will not be replaced to accommodate the new incoming task. Cache partitioning [16] partitions the cache into multiple preserved regions, each of which can only be used by a dedicated task. Obviously, both cache locking and cache partitioning have the drawback that the cache space per task is reduced. Cache-aware execution time analysis [14] improves the precision of worst-case execution time estimation by taking cache effects into the preemption delay calculation. However, these approaches do not address dynamic cache reconfiguration.

Our previous work on cache reconfiguration in real-time systems is presented in [15], which utilized a single level cache subsystem. As embedded system's capability keeps improving nowadays, two-level cache is becoming common in these systems. However, two-level cache hierarchy has a much larger design space than single-level cache since a cross product of two configuration spaces of two cache levels needs to be considered. This may lead to prohibitively long searching time if brute force algorithm is used. We propose three heuristics to tune two levels of caches in an efficient fashion. We also proposed the algorithm to utilize the static profiling information dynamically to tune the cache hierarchy. Our work is based on a preemptive real-time system using EDF scheduling algorithm and periodic tasks with soft time constraints.

3 Background

In our previous work [15], we statically profiled each task and stored the analysis results in a lookup table which is fully utilized at runtime to make reconfiguration decisions. In this section, we summarize the background on configurable cache architecture and our static profiling technique.

3.1 Configurable Cache Architecture

The configurable caches used in our work are based on the architecture described in [17]. The underlying cache architecture contains four separate banks that can operate as four separate ways. Special configuration registers are used to inform the cache tuner – a custom hardware or a lightweight process – to concatenate ways such that the associativity can be altered. The special registers may also be configured to shut down ways to vary the cache size. Similarly, by configuring the fetch unit to fetch cache lines in various lengths, we can adjust the line sizes. We extend the single level configurable cache in [17] to a two-level cache hierarchy by utilizing a level two data cache as a unified cache. Therefore, our target architecture has separate level one caches – instruction level one cache (**IL1**) and data level one cache (**DL1**) – as well as a unified level two cache (**L2**).

3.2 Phase-based Cache Tuning

Research shows that application's operating requirements varies throughout the execution [13]. Hence, the energy savings by tuning configurable parameters for the whole application still has potential for improvement. Since a preemptive system is considered, executing tasks may be interrupted and preempted by newly arrived tasks with higher priorities. Due to this nature, tuning cache hierarchy at the granularity of execution intervals may yield more energy savings and less performance unpredictability.

Within a single task, potentially there exist several intervals of different lengths having distinct operating behaviors. However, it is not feasible to utilize these inborn intervals because preemption could happen at any point throughout the execution. In other words, when a preempted task resumes, the cache requirement of the remaining part may greatly differ from the entire task due to its distinguishing behaviors. So the best option is to use a Monte Carlo style method. As shown in Figure 1, each task is evenly divided with n predefined *potential preemption points*. A *phase* is defined as the execution interval from one partition point to task completion. The number of partition points is defined as *partition factor*. Experiments in

[15] show that a partition factor around four to seven is sufficient to yield the majority of energy savings. Here, C_1, C_2 ... C_n represent the chosen cache configurations for each phase. Note that since we are considering two-level caches, each C_i actually stands for three cache configurations (IL1 cache, DL1 cache and L2 cache).

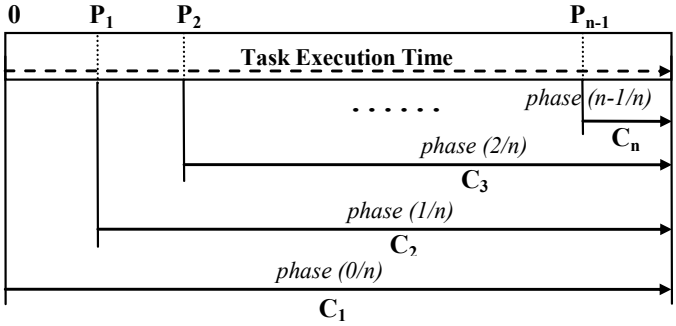

Figure 1. Phase-based cache tuning: task is partitioned at n potential preemption points (P_i) resulting in n phases.

The phase-based profiling generates a *profile table* which stores optimal cache configurations for each phase of a task. For each task, the energy- and performance- optimal cache configurations of all phases are found and stored in the profile table. It also stores the total number of execution cycles required in each phase. Differing from [15], we also take L2 cache also into account. Both energy-optimal configuration for L1 cache (EO_i^{il1}(n/p), EO_i^{dl1}(n/p)) and L2 cache (EO_i^{ul2}(n/p)) of the n^{th} phase of task i are stored. EOT_i(n/p) represents the n^{th} phase's execution time if the caches are tuned to these configurations. Similarly, the same set of information are stored for the performance-optimal cache configurations.

4 Reconfiguration of Two-Level Caches

In this section, we present our work on cache reconfiguration for soft real-time systems with a two-level cache hierarchy. First, we describe how to generate profile table with profitable cache configurations using efficient heuristics. Next, we present an algorithm on how to use the profile table to dynamically reconfigure cache hierarchy.

4.1 Two-Level Cache Exploration

Tuning a two-level cache faces the difficulty of exploring an enormous configuration space. In this paper, we examine typical exploration parameters of conventional embedded systems. We explore cache sizes of 1KB, 2KB and 4KB, line sizes of 16, 32 and 64 bytes, and direct-mapped, 2- and 4-way set associativities for the L1 cache. We use a 4KB cache architecture proposed in [17] with four banks each of which is 1KB. Since the reconfiguration of associativity is achieved by way concatenation as described in Section 3.1, 1KB L1 cache can only be direct-mapped as other three banks are shut down. For the same reason, 2KB cache can only be configured to direct-mapped or 2-way associativity. Therefore, there are 18 (=3+6+9) configuration candidates for L1 caches. Let S_{il1} and S_{dl1} denote the size of exploration space for IL1 cache and DL1 caches, respectively. So we have S_{il1} = 18 and S_{dl1} = 18. For L2 cache, we choose 8KB, 16KB and 32KB as cache sizes; 32, 64 and 128 bytes as line sizes; 4-, 8- and 16-way set associativities with a 32KB cache architecture composed of four separate banks. Similarly, there are 18 possible configurations (S_{ul2} = 18). For comparison, we have chosen a ***base cache*** hierarchy, which reflects a global optimal configuration for all the tasks, consisting of two 2KB, 2-way set associative L1 caches with a 32 byte line size, and a 16KB, 8-way set associative unified L2 cache with a 64 byte line size. The remainder of this section describes our proposed exploration techniques.

4.1.1 Exhaustive Exploration

Intuitively, if the two levels of caches can be explored independently, one can easily profile one level at a time while holding the other level to a typical configuration, which will result in a much small exploration space. However, it is not reasonable to claim that the combination of three independently found energy-optimal configurations actually is or ever close to the global optimal one. The two cache levels affect each other's behavior in various ways. For instance, L2 cache's configuration determines the miss penalty of the L1 caches. Also, the number of L2 cache accesses directly depends on the number of L1 cache misses.

Therefore, the only way to hire the optimal configuration is to search the entire space exhaustively. Since the instruction cache and the data cache could have different configurations, there are 324 (=S_{il1}*S_{dl1}) possible configurations for L1 cache. Addition of the L2 cache increases the design space size to 4752[1]. Moreover, the phase-based profiling strategy we use makes this number even larger. For a single task, if the partition factor is 4, we have to explore for all four phases, leading to a total of 19008 task phase executions. Obviously it is infeasible. We use the exhaustive method for comparison with the heuristics presented in the following sections.

4.1.2 Independent Level One Cache Tuning – IL1T

While different cache levels are dependent on each other, our experimental results demonstrate that instruction cache and data cache are relatively independent. In this study, we fix one's configuration while changing the other's to see whether the varying one has impact on the fixed one. We observe that the profiling statistics for the instruction cache almost remain identical with different data caches and vice versa. It is mainly due to the fact that access pattern of L1 cache is purely determined by the application's characteristics, and the instruction and data streams are relatively independent from each other. Furthermore, factors affecting the instruction cache's energy consumption as well as performance (such as hit energy, miss energy and miss penalty cycles) have very little dependency on the data cache and vice versa.

[1] Not equal to S_{il1}*S_{dl1}*S_{ul2} because the candidates in which L2 cache's line size is smaller than any of the L1 caches are eliminated

This observation offers an opportunity to reduce the exploration space. We propose IL1T – Independent Level One Tuning heuristic – during which IL1 and DL1 caches always use the same configuration while exploring with all L2 cache configurations. This method results in a total of 288 configurations – a considerable cut down of the original quantity, though still not small. Throughout the static analysis, we make book keeping including the energy consumptions and miss cycles of each cache individually. The energy-optimal IL1 cache is the one with the lowest energy consumption of itself (and same for DL1 cache and L2 cache). We choose the cache configuration combination composed of the three locally energy-(performance-) optimal caches as the energy- (performance-) optimal cache hierarchy to be stored in the profile table.

4.1.3 Two-Step Tuning – TST

By examining the results generated by IL1T, we find that some very unprofitable L1 cache configurations are also explored 18 (=S_{ul2}) times with L2 cache, resulting in still relatively inferior cache combinations. These non-beneficial configurations are likely to be discarded. Just like in single level cache tuning, we only have to consider configurations which offer Pareto-optimal tradeoff points. Candidates with both lower performance and higher energy consumption than one of these Pareto-optimal ones are eliminated during exploration. Our proposed Two-Step Tuning heuristic is summarized below:

1. Hold DL1 and L2 as the base cache. Tune IL1 and record all its Pareto-optimal configurations. Let P_{il1} denote the number of recorded IL1 configurations.
2. Hold IL1 and L2 as the base cache. Tune data cache and record all its Pareto-optimal configurations. Let P_{dl1} denote the number of recorded DL1 configurations.
3. Hold both L1 caches as the base cache. Tune L2 and record all its Pareto-optimal configurations. Let P_{ul2} denote the number of recorded L2 configurations.
4. Explore all the combinations from each set of Pareto-optimal configurations recorded in the previous steps and find the energy- and performance- optimal configurations.

The first three steps explore 54 (=S_{il1}+S_{dl1}+S_{ul2}) candidates while the last step explores P_{il1}*P_{dl1}*P_{ul2} candidates. Based on our experimental results, the number of Pareto-optimal points varies from application to application but normally around 3 to 5. Therefore, the total exploration space is reduced to 81 - 179, though in some cases the number could be larger than IL1T's space size (288).

4.1.4 Interlaced Tuning – ILT

Gordon-ross et al. [3] designed a tuning heuristic named TCaT – Two-level Cache Tuning – in a interlaced manner for desktop systems. In their approach, cache parameters are tuned in the order of their importance to the overall energy consumption, which is cache size followed by line size and finally associativity. TCaT claims to find the configuration with energy consumption close to the optimal one by only exploring tens of candidates. We adapt the strategy used in TCaT and propose ILT – Interlaced Tuning heuristic – which finds both energy- and performance- optimal parameters throughout the exploration. In order to increase the chances finding optimal L2 cache size, which we believe has the highest importance, we combine the exploration of L2 cache's size and associativity together. ILT is described below:

1. First, tune by cache size. Hold the IL1's line size, associativity as well as DL1 to the smallest configuration. L2 is set to the base cache. Explore all three instruction cache sizes (1KB, 2KB and 4KB) and find out the energy- and performance- optimal one(s). Perform same explorations for DL1 cache size. In L2 size exploration, we try all the associativities for each cache size and repeat the process twice to find the energy- and performance- optimal size(s), separately. We set L1 sizes to the energy-(performance-) optimal ones in the process of finding energy- (performance-) optimal L2 size(s).
2. Next, tune by line size. We set cache sizes to the energy-(performance-) optimal ones and L2's associativity found in the first step in exploring energy- (performance-) optimal line sizes for each cache. These two tasks are repeated for both L1 caches and L2.
3. Finally, tune by associativity. We set the cache sizes and line sizes to the energy- (performance-) optimal ones in exploring energy- (performance-) optimal associativity. Note that we only explore associativities for L1 caches in this step. During the process of finding DL1's optimal associativities, we already have all the other parameters we needed to compute the total numbers of execution cycles that are required in the profile table.

In the worst case, ILT explores 54 configurations: the first step explores 6 for L1 caches and 18 for L2 cache; the second step explores 18 (=6*3) candidates; final step explores 12 (=6*2) candidates. However, in most cases, there are a lot of repetitive configurations throughout the process that we only have to execute once. In practice, ILT has a exploration space size of around 35 configurations.

4.2 Scheduling-Aware Reconfiguration

This section describes the algorithm we propose to dynamically reconfigure the cache hierarchy at runtime using the static analysis results stored in the profile table. Additionally, as exhibited in Table 1, there is a *task list* that maintains necessary book keeping information for each task. Current Phase (CP_i) denotes the last partition point which the task execution has just passed through. Like common real-time systems, a ready task list (RTL) is also maintained as a priority queue comprising all the tasks ready to execute ordered by priority[2].

Table 1. Task list entry sample.

Task ID: i	Partition Factor: p
Current Phase (CP_i)	Deadline (D_i)
Total Instruction Number (TIN_i)	Executed Instruction Number (EIN_i)

[2]Here the priority means the dynamic scheduling priority decided by EDF.

Algorithm 1 Cache configuration selection

Input: Task list entry, ready task list and profile table.
Output: An appropriate cache configuration combination.
Step 1: Calculate the preempted task T_p's CP.
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: Get the task with maximum priority T_c from RTL.
Step 3: Sort all tasks in RTL by priority, T_1 to T_m, from highest to lowest. t represents the current time instant.
for $j = 1$ to m **do**
  if $t + POT_{Tc}(CP_{Tc}/p) + \sum_{i=1}^{j} POT_{Ti}((CP_{Ti}+1)/p) > D_{Tj}$ **then**
    Task D_{Tj} is subject to be discarded;
  end if
end for
Step 4: Select cache configuration for T_c. Let m′ be the number of tasks in RTL left after **Step 3**.
if $t + EOT_{Tc}(CP_{Tc}/p) > D_{Tc}$ **then**
  $EO\_OK = false$;
else
  $EO\_OK = true$;
  for $j = 1$ to m' **do**
    if $t + EOT_{Tc}(CP_{Tc}/p) + \sum_{i=1}^{j} POT_{Ti}((CP_{Ti}+1)/p) > D_{Tj}$ **then**
      $EO\_OK = false$;
    end if
  end for
end if
if $EO\_OK == true$ **then**
  $Cache_{Tc}^{il1} = EO_{Tc}^{il1}(CP_{Tc}/p)$; $Cache_{Tc}^{dl1} = EO_{Tc}^{dl1}(CP_{Tc}/p)$;
  $Cache_{Tc}^{ul2} = EO_{Tc}^{ul2}(CP_{Tc}/p)$;
else
  $Cache_{Tc}^{il1} = PO_{Tc}^{il1}(CP_{Tc}/p)$; $Cache_{Tc}^{dl1} = PO_{Tc}^{dl1}(CP_{Tc}/p)$;
  $Cache_{Tc}^{ul2} = PO_{Tc}^{ul2}(CP_{Tc}/p)$;
end if
Return: $Cache_{Tc}^{il1}$, $Cache_{Tc}^{dl1}$, $Cache_{Tc}^{ul2}$

Algorithm 1 illustrates cache configuration selection algorithm. This algorithm is called either when a new task with a higher priority than the current executing task arrives in the system or when the current task finishes its execution. In the former case, Step 1 uses the executed instruction number (EIN) to calculate the Current Phase (CP) for the preempted task. In the latter case, this step should be omitted. Step 2 picks the highest priority task T_c from RTL. In the former case, the newly arrived task is inserted into RTL and, obviously, T_c refers to that task. Step 3 checks the schedulability of all 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 the order of tasks' priority (from highest to lowest) to achieve least discarded tasks. Step 3 is skipped if RTL is empty. In Step 4, the appropriate cache configuration for T_c is selected based on whether it is safe to use 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.

5 Experiments

5.1 Experiment Setup

To evaluate our exploration heuristics and scheduling algorithm, we selected six benchmarks from MediaBench [8] and another six benchmarks from EEMBC [5] benchmark suites. These benchmarks are all specially designed for embedded systems and suitable for the cache configuration parameters described in Section 4.1. Table 2 shows our four task sets, each of which consists of three selected benchmarks. In order to avoid the situation where one or two tasks dominate the total energy consumption, tasks in each set are chosen to have comparable sizes. All the tasks are executed with the default input sets provided with the benchmark suites.

Table 2. Benchmark task sets

	Task 1	Task 2	Task 3
Set 1	epic*	pegwit*	rawcaudio*
Set 2	cjpeg*	toast*	mpeg2*
Set 2	A2TIME01**	AIFFTR01**	AIFIRF01**
Set 3	BITMNP01**	IDCTRN01**	RSPEED01**

**MediaBench **EEBMC*

Our energy model is adapted from the one used in [15] and extended to incorporate a unified L2 cache. In order to fill up the energy model with the actual dynamic cache access energy consumption of each configuration, we obtained values using CACTI 4.2 [6] with a 0.18 μm technology. We implemented the energy model and cache tuning heuristics using Perl scripts, which we used to drive the SimpleScalar toolset [2] to do the phase-based task profiling. In order to get the optimal cache configurations for each phase, we utilized checkpointing and fastforwarding capabilities provided in SimpleScalar which allow us to execute specified intervals of a task. Once we have the profile tables for all the tasks, we use an EDF scheduler to simulate the system. The scheduler calls another script which contains the cache configuration selection algorithm (Algorithm 1) to reconfigure the cache.

5.2 Results

We quantify the cache subsystem energy savings using our approach by comparing to the base cache scenario. We use four cache exploration methods – exhaustive, IL1T, TST and ILT – to generate profile tables for all the task sets. Figure 2 presents the total cache hierarchy energy consumption normalized to the base cache for all the four task sets using each exploration technique. As expected, exhaustive exploration generated the highest energy saving (49% on average). IL1T achieves 46% average energy saving which is comparable to the exhaustive method. TST outperforms IL1T in task set 4 but on average saves 43% of the energy consumption, while ILT performs the

worst, though still 32% energy saving is achieved. Figure 2 also shows the relative performances of each heuristic. On an average, IL1T, TST, ILT make the system consume 7%, 13% and 34% more energy than the exhaustive method. The reason for IL1T not finding the optimal configurations is that though L1 caches are relatively independent, they both have impact on the L2 cache which has effect back on L1 caches. So they are essentially indirectly dependent on each other through the L2 cache. TST only considers Pareto-optimal configurations at the cost of losing the chance of finding more efficient cache combinations which actually consists of non-beneficial ones. One of the reason is that a less energy efficient (due to oversize) L1 cache may cause less accesses to L2 cache. Hence an appropriate L2 cache may make this non-beneficial L1 cache overall better. ILT behaves worst due to the fact that it could miss the optimal parameter easily when exploring with other unknown but randomly chosen parameters.

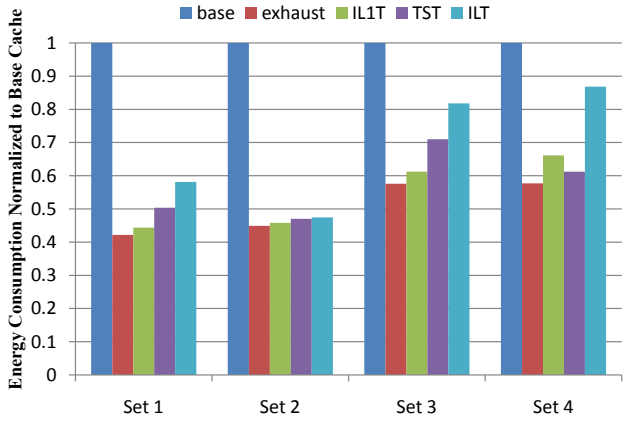

Figure 2. Cache hierarchy energy consumption using four heuristics.

The three heuristics, though exhibits less energy savings, are much more efficient than exhaustive method in the static profiling stage. Table 3 presents the total number of cache configurations explored by each exploration heurstics[3]. Our experience is that it normally takes days to profile a task using exhaustive method while few minutes if ILT is employed. Designers can decide which heuristic to use based on the static profiling time and the overall energy savings.

Table 3. Cache hierarchy configuration explored using different exploration methods.

	Set 1	Set 2	Set 3	Set 4
Exhaust	14256	14256	14256	14256
IL1T	864	864	864	864
TST	419	334	368	403
ILT	106	98	107	104

6 Conclusions

Dynamic reconfiguration techniques are widely used in designing efficient embedded systems. Dynamic cache reconfiguration is a promising approach to improve both energy efficiency and overall performance. In this paper, we present a novel methodology to apply a two-level configurable cache hierarchy in soft real-time systems. Our methodology employs an efficient combination of static analysis and dynamic tuning of cache parameters with very minor impact on timing constraints. Three cache exploration heuristics, which greatly improve the static analysis efficiency, are designed and compared with the exhaustive method. Our results show that up to 46% energy of the cache hierarchy can be saved using our approach.

[3]For simplicity, these numbers only count for the three tasks on the whole in each set but not for every phase.

References

[1] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. *Micro*, 1999.

[2] The SimpleScalar toolset. *http://www.simplescalar.com/*.

[3] A. Gordon-ross and F. Vahid. Automatic tuning of two-level caches to embedded applications. *DATE*, 2004.

[4] A. Gordon-ross and F. Vahid. Fast configurable-cache tuning with a unified second-level cache. *ISLPED*, 2005.

[5] EEMBC, The Embedded Microprocessor Benchmark Consortium. *http://www.eembc.org/*.

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

[7] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[8] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. *Micro*, 1997.

[9] A. Malik, B. Moyer, and D. Cermak. A low power unified cache architecture providing power and performance flexibility. *ISLPED*, 2000.

[10] M. Modarressi, S. Hessabi, and M. Goudarzi. A reconfigurable cache architecture for object-oriented embedded systems. *CCECE*, 2006.

[11] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. *RTSS*, 2002.

[12] A. Settle, D. Connors, and E. Gibert. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 2006.

[13] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *Micro*, 2003.

[14] Y. Tan and V. J. Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Transactions on Embedded Computing Systems*, 2007.

[15] W. Wang, P. Mishra, and A. Gordon-Ross. Sacr: Scheduling-aware cache reconfiguration for real-time embedded systems. *VLSI Design*, 2009.

[16] A. Wolfe. Software-based cache partitioning for real-time applications. *IWRCS*, 1993.

[17] C. Zhang, F. Vahid, and R. Lysecky. A self-tuning cache architecture for embedded systems. *DATE*, 2004.