Dynamic Reconfiguration of Two-Level Cache Hierarchy in Real-Time Embedded Systems*

Weixun Wang and Prabhat Mishra

Department of Computer and Information Science and Engineering

University of Florida

Gainesville, FL, 32601, USA

{wewang, prabhat}@cise.ufl.edu

Address:

CSE 577, University of Florida

Department of Computer and Information Science and Engineering

Gainesville, FL, 32601, USA

Email: wewang@cise.ufl.edu

Date of Receiving: **to be completed by the Editor**

Date of Acceptance: **to be completed by the Editor**

***** This work was partially supported by NSF grant CCF-0903430 and SRC grant 2009-HJ-1979.

Dynamic Reconfiguration of Two-Level Cache Hierarchy in Real-Time Embedded Systems

Weixun Wang and Prabhat Mishra

Abstract *— System optimization techniques based on efficient dynamic reconfiguration have been widely adopted in recent years. 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 and embedded 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 real-time systems with a unified two-level cache hierarchy. We propose a set of exploration heuristics for our static analysis which effectively reduces the exploration time while keeps the generated profile results beneficial to be leveraged during runtime. Our experimental results have demonstrated 40 - 58% energy savings with minor impact on performance.*

Keywords — Low-power, real-time systems, embedded systems, cache, memory

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 are proposed, targeting at different system components and design levels, by changing tunable system parameters during runtime. These dynamic reconfiguration techniques offer the ability to meet each application's unique requirement. They mainly focus on determining *when* and *how* to reconfigure the system to achieve higher energy efficiency and performance. Memory subsystem nowadays is responsible for as much as 50% of the energy consumption of a microprocessor system [2] [3]. Cache hierarchy, which has much higher access frequency and made of more power expensive SRAM than main memory, occupies the majority part of the memory subsystem’s energy consumption [35]. According to Amdahl's law, such a large contribution to the overall energy consumption makes cache a good candidate for optimization. Cache hierarchy reconfiguration could lead to significant energy saving by meeting application's diverse cache requirements [4] [5]. Specifically, 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 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 the system's service quality. 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. Periodic tasks normally have known characteristics, including worst case execution time (WCET), period and deadline, before execution. Earliest Deadline First (EDF) and Rate Monotonic (RM) [6] are the two most frequently referenced fundamental scheduling algorithms for periodic task sets in real-time system research community. For sporadic and aperiodic task sets, each task's information is not fully known a priori and, therefore, can arrive at any time instance. Sporadic tasks usually have hard time constraints and are accepted into the system only if they pass the schedulability test when they arrive. On the other hand, aperiodic tasks are scheduled whenever there is enough slack time. Hence, aperiodic tasks normally have soft deadlines and can only be scheduled as soon as possible. Scheduling algorithms for sporadic and aperiodic tasks can be found in [7] [6] [8].

While dynamically reconfigurable cache has been well studied for desktop as well as traditional embedded systems [9] [10], it is still a major challenge to employ cache reconfiguration in real-time systems. Both determining the appropriate configuration and tuning the cache hierarchy 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. Soft real-time system provides unique flexibility to utilize cache reconfiguration to exploit considerable energy saving at the cost of minor impacts on service levels. Our previous work [11] has 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. In this paper, we efficiently employ 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 0 presents our experimental results. Finally, Section 6 concludes the paper.

2 RELATED WORK

2.1 Energy-aware Real-Time Scheduling Techniques

Dynamic Power Management (DPM) [12] and Dynamic Voltage Scaling (DVS) [13] are the most prominent techniques used in energy-aware scheduling for real-time systems. DPM takes advantage of the processor idle time (slack time) to reduce the overall energy consumption by putting the system into an ultra-low power sleep mode. DVS methods, on the other hand, can be employed to achieve the same goal by adjusting the processor voltage level (along with operating frequency) at runtime [14] [15] [16] [17] [18]. The reason behind DVS's ability to save energy consumption is that lowering voltage level will lead to quadratic power consumption reduction but nearly linear performance slow down. In both cases, timing constraints of tasks (e.g., deadlines) must be considered during decision making. A survey on energy-aware scheduling techniques can be found in [19]. Unfortunately, none of them considers cache reconfiguration. Our proposed approach is complementary to any of these techniques.

2.2 Caches in 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 has been carried out to employ caches in real-time systems. Cache-aware WCET analysis predicts the impact on task execution time from cache behaviors during design time [20]. Puant et al. [21] 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 [22] 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 [23] [24] 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.

2.3 Dynamic Cache Reconfiguration

Many general and application specific reconfigurable cache architectures have been proposed over the years. Motorola M*CORE processor [2] 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. [25] developed a cache architecture which can be dynamically partitioned and resized to improve the performance of object-oriented embedded systems. Settle et al. [26] proposed a dynamically reconfigurable cache specifically designed for chip multi-processors. Zhang et al. [27] proposed an efficient and highly configurable cache architecture which imposes almost no overhead to the critical path.

A lot of research efforts are spent in finding efficient automated techniques to reconfigure the cache hierarchy. Dynamic and static analyses 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. Gordon-Ross et al. [28] proposed a non-intrusive, N-experts based analysis technique in which an auxiliary structure is used to evaluate all cache configurations simultaneously. However, the auxiliary data structure is too power expensive and thus not applicable in real-time embedded systems, especially when multi-level cache hierarchy is considered. 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 whole application or a part of it. The former strategy is called application-based tuning [10] while the latter is called phase-based tuning [28] [29]. Previous works for tuning two-level cache hierarchy focused on design space reduction in desktop-based systems. Exploration heuristics introduced in [9] and [10] are designed for a configurable cache hierarchy with separate level-one caches [2] and with a unified level-two cache, respectively. However, none of these works is designed for systems with real-time constraints. Furthermore, existing exploration heuristics are not enough to make flexible tradeoff between running time and solution quality.

Our previous work on cache reconfiguration in real-time systems is presented in [11], which utilized a single level cache subsystem. As embedded system's capability keeps improving nowadays, two-level cache is becoming common. 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 four heuristics to tune two levels of caches in an efficient fashion. We also propose the algorithm to utilize the static profiling information dynamically to tune the cache hierarchy. Our work is based on soft real-time systems with preemptive scheduling. Both periodic and aperiodic task sets are applicable as long as static profiling can be effectively carried out for each task.

3 BACKGROUND

In our previous work [11], 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 [27]. As shown

in Figure 1 (a), the underlying cache architecture consists of four separate banks where each of them acts as a separate way. The cache tuner can be implemented either as a small custom hardware or lightweight software running on a co-processor which changes the cache configuration through special registers. In order to enable associativity reconfigurability, way concatenation, shown in Figure 1 (b), can logically concatenate ways together. Varying cache size is achieved by shutting down certain ways as shown in Figure 1 (c). Cache line size is configured by setting a unit-length base line size and then fetching subsequent lines if the line size increases as illustrated in Figure 1 (d).

We extend the single level configurable cache in [27] 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

Earlier works have found that different application have distinct favored L1 cache configurations [5] [27] [30]. However, as shown in Table 1, optimal L2 cache configurations (both in terms of energy and performance) vary among different applications. For example, a 8KB L2 cache with 4-way associativity and 32-byte line size is sufficient for exploiting *cjpeg*'s locality and results in the minimum amount of energy consumption. However, for *epic*, the L2 cache configuration with 16KB capacity turns out to be most energy efficient since the miss rate is too high for 8KB L2 cache[1]. Clearly, by dynamically changing the configuration of the cache hierarchy, we can satisfy each task's requirement and therefore achieve system optimization goals.

Research also shows that application's operating requirements varies throughout the execution [31]. Hence, the energy savings by tuning configurable parameters for the whole application still has potential for improvement. Since a preemptive real-time system is considered, executing tasks may

[1] For ease of discussion, we use the dotation “XXKB_XXW_XXB” to represent the cache configuration with XX KB capacity, XX-way associativity and XX bytes line size.

be interrupted and preempted at any time by newly arrived tasks with higher priorities. Due to this nature, tuning the 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 intrinsic 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 2, 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 our earlier work [11] 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).

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 [11], we also take L2 cache 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 is stored for the performance-optimal cache configurations. Table 2 shows an example of a profile table.

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 Design Space Exploration for Reconfigurable Two-Level Caches

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 associativity for the L1 cache. We use a 4KB cache architecture proposed in [5] 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 simplicity, which is also practically true in most scenarios, IL1 and DL1 has the same exploration space which is denoted by S_{l1}. 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 associativity 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 fixed configuration for all the tasks if cache reconfiguration is not available, consisting of two 2KB, 2-way set associative L1 caches with a 32 byte line size (2KB_2W_32B), and a 16KB, 8-way set associative unified L2 cache with a 64 byte line size (16KB_8W_64B). 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 obtain 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[2] to 4752. Moreover, the phase-based static 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 Same Level One Cache Tuning – SLOT

As discussed above, the design space explosion is resulted from the cross-product of three separate design spaces: IL1, DL1 and L2. The most straightforward optimization is to remove one dimension (i.e., space) so that the total exploration time is drastically reduced while the solution quality is mostly preserved. Our studies show that, for many real applications, the favored (both in terms of energy efficiency and performance) IL1 and DL1 cache configurations are similar to each other (at least in cache size).

[2] Not equal to S_il1* S_dl1* S_ul2 because candidates whose L2 cache's line size is smaller than L1 are eliminated.

Therefore, we propose SLOT -- Same Level One Cache 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 (94%) of the original quantity (4752), though still not small enough.

4.1.3 Two-Step Tuning – TST

By examining the results generated by SLOT, we find that some very unprofitable L1 cache configurations are also explored 18 (= S_{ul2}) times with L2 cache, resulting in still relatively inferior energy efficiency and performance when combined together as the cache hierarchy configuration. These non-beneficial configurations are likely to be discarded. Therefore, just like in single level cache tuning, we only have to consider configurations which offer Pareto-optimal tradeoff points. In other words, for each individual cache, candidates which have both lower performance and higher energy consumption than any other one(s) can be safely eliminated during exploration. Then, the design space which contains the cross-product of all three sets of Pareto-optimal points is explored. Our proposed Two-Step Tuning (TST) heuristic is summarized below:

- 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.
- 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.
- 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.
- Explore all the combinations from each set of Pareto-optimal configurations recorded in the previous steps and find the energy- and performance- optimal cache hierarchy 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 configurations varies from application to application but normally around 3 to 5. Therefore, the total exploration space is reduced to roughly 81 - 179 (a reduction of 38% to 72%), though in some worst cases the number could be larger than SLOT's space size (288).

4.1.4 Independent Level One Cache Tuning – ILOT

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.

This observation offers an opportunity to further reduce the exploration space. We can use the same configurations for IL1 and DL1 while L2 is fixed to base cache to find the "local optimal" configurations for L1 caches. Specifically, throughout the static analysis, we record the energy consumptions and miss cycles of each cache individually. The local energy-optimal IL1 cache is the one with the lowest energy consumption of itself (and same for DL1 cache and L2 cache). The local performance-optimal cache is determined by the number of miss cycles for each cache. ILOT is summarized as below:

- Hold L2 as the base cache. Explore all L1 cache configurations during which IL1 and DL1 are always configured to the same configuration. Local optimal (both energy- and performance-) configurations for both IL1 and DL1 are recorded.

- Hold IL1 and DL1 as the energy-optimal configurations found in the last step. Explore all L2 cache configurations and record local energy-optimal L2 cache configuration. The process is repeated for performance-optimal L2 configuration also.
- The energy- (performance-) optimal configuration for the cache hierarchy is composed of the three local energy- (performance-) optimal caches for each separate cache.

Clearly, the first step simulates 18 (= S_{l1}) configurations while the second step requires 36 (= S_{ul2} * 2) explorations. If some local optimal IL1 and DL1 configurations happen to be identical, the second step may take less number of explorations. The last step potentially takes 2 simulations. In total, discarding repeating configurations, ILOT has an exploration space of no more than 54 configurations.

4.1.5 Interlaced Tuning – ILT

Gordon-ross et al. [9] designed a tuning heuristic named TCaT -- Two-level Cache Tuning -- in a interlaced manner for desktop systems with unified level one and level two caches. 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. Therefore, as opposed to [9], in each step other than the first, we need to set the already-explored parameters to energy- and performance-optimal ones separately during the exploration of the current parameter. In order to increase the chances of finding optimal L2 cache size, which we found has the highest importance, we combine the exploration of L2 cache's size and associativity together. We sacrifice a certain amount of exploration time for better profiling results. ILT is summarized as below:

- 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). Same explorations are performed for DL1 cache size. In L2 size exploration, we try all the associativities (4W, 8W and 16W) with each L2 cache size (8KB, 16KB and 32KB) 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 corresponding process of finding energy- (performance-) optimal L2 size(s).

- Next, tune by line size. We set the cache sizes and L2 cache's associativity to the energy- (performance-) optimal ones found in the first step during exploring energy- (performance-) optimal line sizes for each cache (16B, 32B and 64B for L1 caches while 32B, 64B and 128B for L2 cache), respectively. These two tasks are repeated for both L1 caches and L2.

- Finally, tune by associativity. We set the cache sizes and line sizes to the energy- and performance-optimal ones when we explore for the energy- and performance-optimal associativity (1W, 2W and 4W), respectively. 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 need to compute the total numbers of execution cycles that are required in the profile table.

At the beginning, we do not have any explored parameter so the L1 cache size tuning is done in one-shot for both IL1 and DL1, which lead to 6 (= 3 + 3) configurations. During L2 cache size tuning, there are 9 (= 3 * 3) possible combinations with the associativity and the process has to be done twice for both energy- and performance- optimal L1 cache sizes. Hence, the first step requires to explore 24 (= 6 + 9 * 2) configurations. Similarly, the second step explores all three lines sizes for each cache separately twice which leads to 18 (= 3 * 2 * 3) candidates. The final step explores 12 (= 3 * 2 * 2) configurations since L2 associativity has already been examined in the first step. Therefore, in the

worst case, ILT explores 54 (= 24 + 18 + 12) configurations. However, in most cases, we observe that there are a lot of repetitive configurations throughout the process which we only have to profile once. For example, the L1 configuration 2KB_1W_16B in the second step has already been explored in the first step. Furthermore, all the configurations composed of invalid cache parameter combinations are also discarded. 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 reconfigure the cache hierarchy at runtime using the static analysis results stored in the profile table. Additionally, as exhibited in Table 3, there is a *task list* that maintains necessary book keeping information for each task instance. 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[3].

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 other words, this procedure decides the cache hierarchy configuration whenever there is a context switch. In the former case, Step 1 first uses the executed instruction number (EIN) to calculate the Current Phase (CP) for the preempted task. This information is stored in the preempted task's list entry and is used by the algorithm when it gets resumed. The ready-to-execute (i.e. current) task T_c is obviously the preempting task. In the latter case, T_c is the one with the highest scheduling priority in RTL. Step 2 checks the schedulability of all the task instances 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 2 is

[3] Here the priority means the dynamic scheduling priority decided by the scheduler.

skipped if RTL is empty. In Step 3, the appropriate cache configuration for T_c is selected based on whether it is safe to use the energy-optimal one. Specifically, it is unsafe (and performance-optimal configuration will be used) when using energy-optimal configuration will violate T_c's own deadline or any other deadline of the tasks left in RTL if they all use the performance-optimal configurations. Note that the next incoming phase $CP_{T_i} + 1$ is used in the time estimation for other tasks in RTL. It is an underestimation and thus T_c may have more chance to select the energy-optimal cache. 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. Obviously, it is efficient enough to be executed at runtime.

5 EXPERIMENTS

5.1 Experiment Setup

To evaluate our exploration heuristics and scheduling algorithm, we selected six benchmarks from each of the following benchmark suits: MediaBench [32] (*cjpeg, epic, pegwit, rawcaudio, mpeg2, toast*), MiBench [33] (*CRC32, dijkstra, FFT, pktflow, qsort, rijndael, susan*) and EEMBC [34] (*A2TIME01, AIFFTR01, AIFIRF01, BITMNP01, IDCTRN01, RSPEED01*). These benchmarks are all specially designed for embedded systems and suitable for the cache configuration parameters described in Section 4.1. Table 4 shows our seven task sets, each of which consists of six selected benchmarks. Task set 1 consists of benchmarks from MediaBench, set 2 from MiBench, set 3 from EEMBC and set 4 - 7 are mixtures from all threes suites. 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.

Our energy model is adapted from the one used in [11] 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 [35] 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 [36] 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 a task 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

5.2.1 Optimal Cache Configuration Selection

First we evaluate our proposed design space exploration heuristics by comparing the energy-(performance-) optimal cache configurations found using each heuristic to the exhaustive approach. This comparison directly reflects the effectiveness of each heuristic (the closer to the exhaustive approach the better). Since these design space exploration results are used to construct the profile table, it will have impact on the scheduling-aware reconfiguration algorithm.

Figure 3 and 4 show the heuristic searching results for selected benchmarks. From Figure 3, we can observe that, for most of the time, all four heuristics behaves well in finding energy-optimal cache hierarchy configurations. For example, for benchmark *dijkstra*, *cjpeg*, *rawcaudio* and *RSPEED01*, all four heuristics are able to find configurations which are very close to the optimal. However, in certain cases, some heuristics may lead to inferior exploration results. For example, both ILOT and ILT do not work well for *pegwit*. Generally speaking, with respective to energy consumption, SLOT and TST behave consistently well among all benchmarks. ILOT behaves very close to TST, sometimes even better (e.g., *cjpeg*, *AIFIRF01*), but could be inferior in other cases. ILT, though having the smallest exploration space and thus being fastest, is only able to find the optimal configurations with the quality 30% away from the optimal on average.

Figure 4 shows the exploration results in terms of performance. In other words, the execution time of the performance-optimal cache configuration found by each heuristic is compared with the

exhaustive search. It can be observed that SLOT and TST are able to consistently find the actual performance-optimal configurations or at least very close ones. On the other hand, although behaves very well in terms of energy consumption, ILOT is not good at finding the performance-optimal configuration for a number of benchmarks. In this aspect, ILT outperforms ILOT.

5.2.2 Energy Saving

We quantify the cache subsystem energy savings using our approach by comparing to the base cache scenario. We use five cache exploration methods -- exhaustive, SLOT, TST, ILOT and ILT -- to generate profile tables for all the task sets. Figure 5 presents the total cache hierarchy energy consumption normalized to the base cache for all the seven task sets using each exploration technique. As expected, exhaustive exploration generated the highest energy saving (58% on average). SLOT achieves 56% average energy saving which is comparable to the exhaustive approach. TST outperforms SLOT in some task sets but on average saves 52% of the energy consumption. While ILOT and ILT perform the worst, we can still achieve 46% and 40% of energy savings, respectively. Figure 5 also shows the relative comparison of each heuristic. On an average, SLOT, TST, ILOT and ILT make the system consume 2.8%, 9.1%, 25.6% and 43.1% more energy than the exhaustive method.

5.2.3 Insights behind Results

It is helpful to examine some insights behind the results shown above. SLOT simply discards the flexibility and benefit of running IL1 and DL1 cache separately. Therefore, when optimal configurations for IL1 and DL1 are different, SLOT will have to suffer from decreased energy efficiency and/or performance in either IL1 or DL1. 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. Specifically, when searching for the Pareto-optimal points for each cache, the other two caches are fixed to the base case. In other words, it is assumed that the Pareto-optimal configuration set for each individual cache is independent of the other cache's

configuration. However, the assumption does not always hold. One of the reasons is that a less energy efficient (due to oversize) L1 cache may cause fewer accesses to L2 cache. Hence an appropriate L2 cache may make this non-beneficial L1 cache overall better. The reason for ILOT not finding the optimal configurations is that, although relatively independent from each other, IL1 and DL1 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. Furthermore, varying one of them, say DL1, will lead to different total execution time and thus the static power consumption of the other (IL1) is also going to change. Therefore, although miss rate is unaffected, IL1 and DL1 do have impact on each other in terms of energy consumption as well as performance. ILT behaves worst due to the fact that it could miss the optimal parameter easily when exploring with other unknown but fixed parameters.

5.2.4 Exploration Efficiency

The four heuristics, though exhibits less energy savings, are much more efficient than exhaustive method in the static profiling stage. Table 5 presents the total number of cache configurations explored by each exploration heuristics[4] for each benchmark. Our experience is that it normally takes days to profile a task using exhaustive method while a few minutes if ILT is employed. For example, exhaustive exploration of all configurations for *qsort* takes about 5 days and 16 hours while only 44 minutes are required for ILT heuristic. Our approach would be more valuable in multi-core scenarios where the design space is even larger. In general, designers can decide which heuristic to use based on the profiling time they have and the expected overall energy savings. For example, if 20 hours per task is permitted, SLOT should be adopted for best energy savings. If design time is limited, ILOT or ILT could be used for best performance.

6 CONCLUSIONS

Dynamic reconfiguration techniques are widely used in designing efficient embedded systems.

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

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. Four cache exploration heuristics, which greatly improve the static analysis efficiency, are designed and compared with the exhaustive method. Our results show that up to 40 - 58% energy of the cache hierarchy can be saved using our approach.

REFERENCES

[1] W. Wang and P. Mishra, "Dynamic reconfiguration of two-level caches in soft real-time embedded systems," Proceedings of IEEE Computer Soceity Annual Symposium on VLSI, (**2009**), pp. 145-150.

[2] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," Proceedings of International Symposium on Low Power Electronics and Design, (**2000**), pp. 241-243.

[3] S. Segars, "Low power design techniques for microprocessors," Proceedings of International Solid State Circuit Conference, (**2001**).

[4] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," Proceedings of International Symposium on Microarchitecture, (**1999**), pp. 248-259.

[5] C. Zhang, F. Vahid, and R. Lysecky, "A self-tuning cache architecture for embedded systems," Proceedings of Design, Automation and Test Conference in Europe, (**2004**), pp. 10142.

[6] J. Liu, Real-Time Systems, Prentice Hall, (**2000**).

[7] B. Sprunt, Aperiodic task scheduling for real-time systems, Ph.D. dissertation, Carnegie Mellon University, (**1990**).

[8] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling arbitrary-deadline sporadic task systems on multiprocessors," Proceedings of Real-Time Systems Symposium, (**2008**), pp. 385-394.

[9] A. Gordon-Ross and F. Vahid, "Automatic tuning of two-level caches to embedded applications," Proceedings of Design, Automation and Test Conference in Europe, (**2004**), pp. 208-213.

[10] A. Gordon-Ross, F. Vahid, and N. Dutt, "Fast configurable-cache tuning with a unified second-level cache," Proceedings of International Symposium on Low Power Electronics and Design, (**2005**), pp. 323-326.

[11] W. Wang, P. Mishra and A. Gordon-Ross, "SACR: Scheduling-aware cache reconfiguration for real-time embedded systems," Proceedings of International Conference on VLSI Design, (**2009**), pp. 547-552.

[12] L. Benini, R. Bogliolo, and G. D. Micheli, "A survey of design techniques for system-level dynamic power management," IEEE Transactions on VLSI Systems, (**2000**), Vol. 8, pp.299-316.

[13] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable-voltage core-based systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, (**1999**), Vol. 18, pp.1702-1714.

[14] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez, "Power-aware scheduling for periodic real-time tasks," IEEE Transactions on Computers, (**2004**), Vol. 53, N °5, pp. 584-600.

[15] J. Chen, T. Kuo, and C. Shih, "1 + ε approximation clock rate assignment for periodic real-time tasks on a voltage-scaling processor," Proceedings of International Conference on Embedded Software, (**2005**), pp. 247-250.

[16] R. Jejurikar and R. Gupta, "Energy-aware task scheduling with task synchronization for embedded real-time systems," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, (**2006**), Vol. 25, pp.1024-1037.

[17] G. Quan and X. S. Hu, "Energy efficient dvs schedule for fixed-priority real-time systems," ACM Transactions on Design Automation of Electronic Systems, (**2007**), Vol. 6, pp.1-30.

[18] W. Wang and P. Mishra, "PreDVS: Preemptive dynamic voltage scaling for real-time systems using approximation scheme," Proceedings of Design Automation Conference, (**2010**).

[19] J. Chen and C. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms," Proceedings of International Conference on Embedded and Real-Time Computing Systems and Applications, (**2007**), pp. 28-38.

[20] I. Puant, "Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems," Proceedings of International Workshop on worst-case execution time analysis, (**2002**).

[21] I. Puant and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," Proceedings of IEEE Real-Time Systems Symposium, (**2002**), pp. 114-125.

[22] A. Wolfe, "Software-based cache partitioning for real-time applications," Proceedings of International Workshop on Responsive Computer Systems, (**1993**).

[23] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," Proceedings of Euromicro Conference on Real-Time Systems, (**2005**), pp. 41-48.

[24] Y. Tan and V. J. Mooney, "Timing analysis for preemptive multitasking real-time systems with caches," ACM Transactions on Embedded Computing Systems, (**2007**), Vol. 6, N °1.

[25] M. Modarressi, S. Hessabi, and M. Goudarzi, "A reconfigurable cache architecture for object-oriented embedded systems," Proceedings of Canadian Conference on Electrical and Computer Engineering, (**2006**), pp. 959-962.

[26] A. Settle, D. Connors, and E. Gibert, "A dynamically reconfigurable cache for multithreaded processors," Journal of Embedded Computing, (**2006**), Vol. 2, pp. 221-233.

[27] C. Zhang, F. Vahid, and W. Najjar, “A highly configurable cache for low energy embedded systems,” CM Transactions on Embedded Computing Systems, (**2005**), Vol. 6, pp. 362-387.

[28] A. Gordon-Ross and F. Vahid, “A self-tuning configurable cache,” Proceedings of Design Automation Conference, (**2007**), pp. 234-237.

[29] A. Gordon-Ross, J. Lau, and B. Calder, “Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy,” Proceedings of the 18th ACM Great Lakes symposium on VLSI, (**2008**), pp. 379-382.

[30] A. Gordon-Ross, P. Viana, F. Vahid, W. Najjar, and E. Barros, “A one-shot configurable-cache tuner for improved energy and performance,” Proceedings of Design, Automation and Test Conference in Europe, (**2007**), pp. 755-760.

[31] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, “Discovering and exploiting program phases,” Proceedings of International Symposium on Microarchitecture, (**2003**), pp. 84-93.

[32] C. Lee, M. Potkonjak, and W. H. Mangione-smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” Proceedings of International Symposium on Microarchitecture, (**1997**), pp. 330-335.

[33] M. Guthaus, J. Ringenberg, D.Ernest, T. Austin, T. Mudge, and R. Brown, “Mibench: A free, commercially representative embedded benchmark suite,” Proceedings of IEEE International Workshop on Workload Characterization, (**2001**), pp. 3-14.

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

[35] CACTI 5.3, HP Labs, http://www.hpl.hp.com/.

[36] D. Burger, T. M. Austin, and S. Bennett, " Evaluating future microprocessors: The simplescalar tool set,” University of Wisconsin-Madison, Tech. Rep. (**1996**).

FIGURES AND TABLES

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

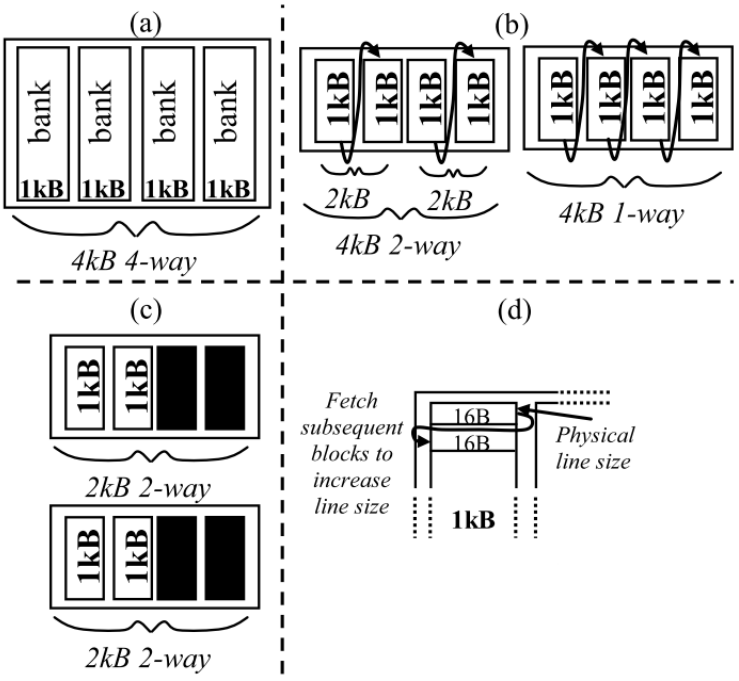

Figure 2 Phase-based cache tuning: task is partitioned at *n* potential preemption points P_i resulting in *n* phases.

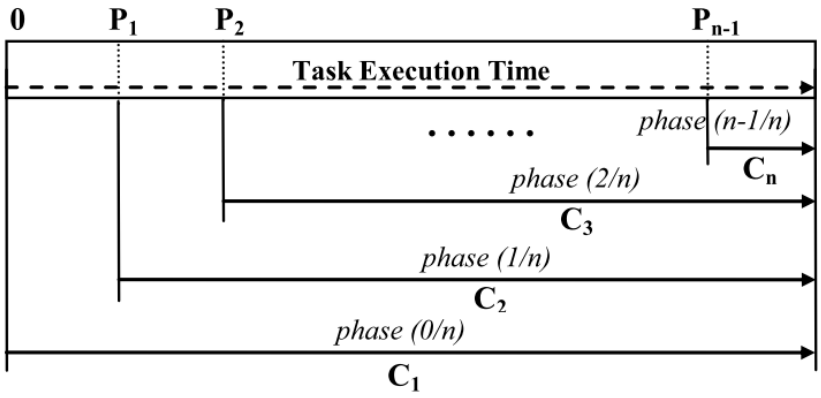

Figure 3 Normalized energy consumption of the searched energy-optimal cache configuration using heuristics.

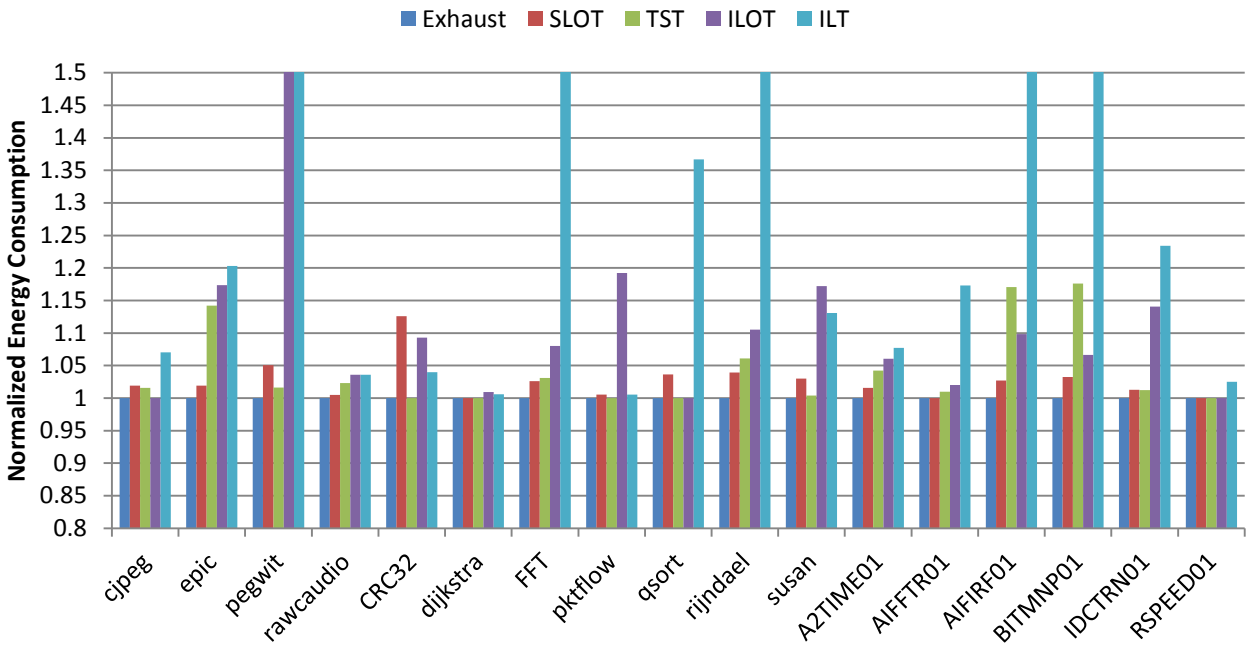

Figure 4 Normalized execution time of the searched performance-optimal cache configuration using heuristics.

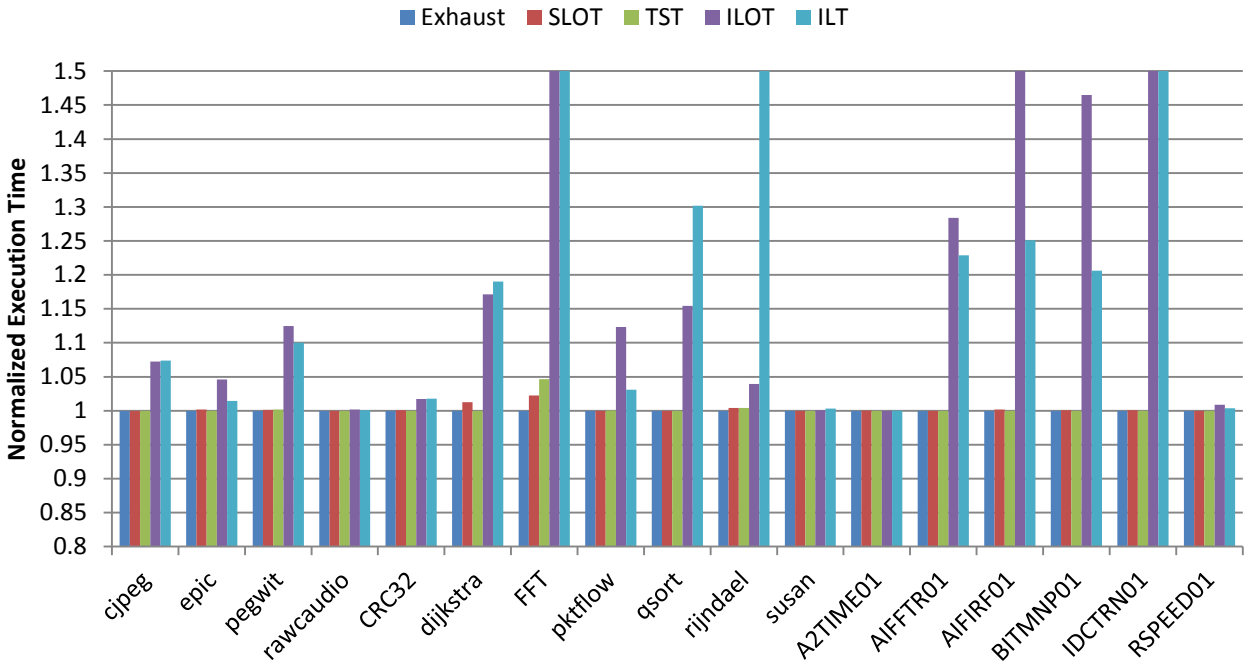

Figure 5 Cache hierarchy energy consumption using four heuristics.

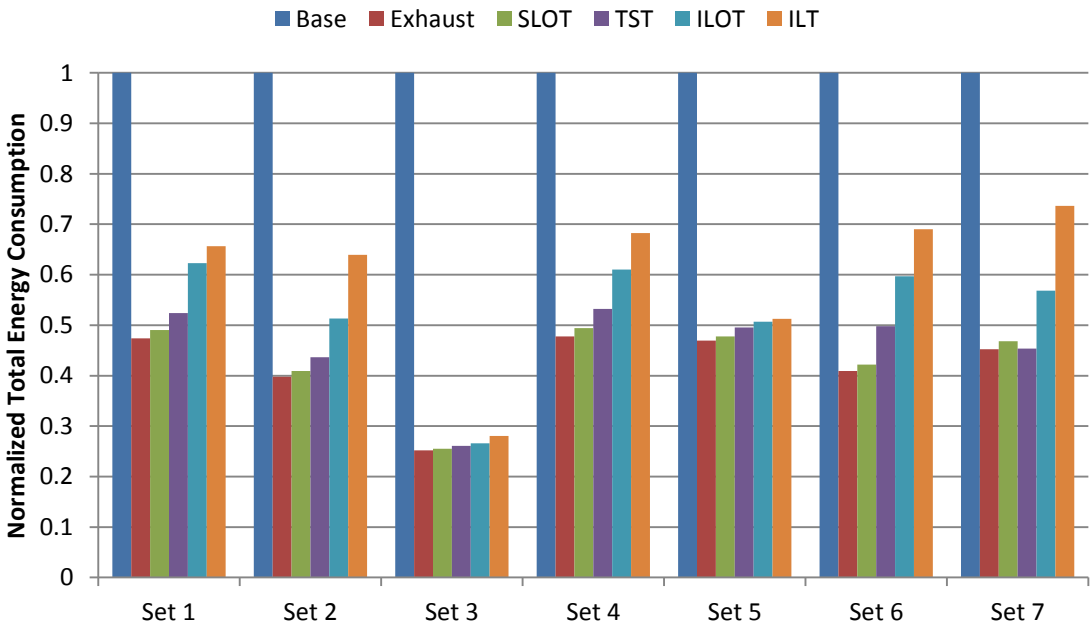

Table 1 Energy optimal (EO) and performance optimal (PO) cache hierarchy configurations for different applications. Each configuration is denoted by its total capacity in kilobytes (KB), followed by the associativity in number of ways (W), followed by the line size in bytes (B).

		IL1	DL1	UL2
cjpeg	EO	4KB_2W_16B	4KB_1W_16B	16KB_4W_32B
	PO	4KB_4W_64B	4KB_4W_64B	32KB_16W_64B
epic	EO	2KB_2W_16B	4KB_1W_16B	16KB_4W_32B
	PO	4KB_2W_64B	4KB_1W_32B	32KB_4W_64B
pegwit	EO	4KB_1W_32B	1KB_1W_16B	8KB_4W_32B
	PO	4KB_2W_32B	4KB_4W_16B	32KB_4W_32B
rawcaudio	EO	1KB_1W_16B	2KB_1W_16B	8KB_4W_32B
	PO	4KB_4W_64B	4KB_2W_32B	32KB_8W_128B
AIFFTR01	EO	4KB_2W_16B	4KB_2W_32B	16KB_4W_32B
	PO	4KB_4W_64B	4KB_2W_64B	32KB_16W_64B
RSPEED01	EO	4KB_1W_32B	2KB_2W_16B	16KB_4W_32B
	PO	4KB_2W_64B	2KB_2W_16B	32KB_16W_64B

Table 2 Static profile table of task i with a partition factor p.

Task i	Partition Factor: p
phase 0	$EO_i^{il1}(0/p)$, $EO_i^{dl1}(0/p)$, $EO_i^{ul2}(0/p)$, $EOT_i(0/p)$ $PO_i^{il1}(0/p)$, $PO_i^{dl1}(0/p)$, $PO_i^{ul2}(0/p)$, $POT_i(0/p)$
phase 1	$EO_i^{il1}(1/p)$, $EO_i^{dl1}(1/p)$, $EO_i^{ul2}(1/p)$, $EOT_i(1/p)$ $PO_i^{il1}(1/p)$, $PO_i^{dl1}(1/p)$, $PO_i^{ul2}(1/p)$, $POT_i(1/p)$
phase 2	$EO_i^{il1}(2/p)$, $EO_i^{dl1}(2/p)$, $EO_i^{ul2}(2/p)$, $EOT_i(2/p)$ $PO_i^{il1}(2/p)$, $PO_i^{dl1}(2/p)$, $PO_i^{ul2}(2/p)$, $POT_i(2/p)$
.....	
phase p-1	$EO_i^{il1}(p-1/p)$, $EO_i^{dl1}(p-1/p)$, $EO_i^{ul2}(p-1/p)$, $EOT_i(p-1/p)$ $PO_i^{il1}(p-1/p)$, $PO_i^{dl1}(p-1/p)$, $PO_i^{ul2}(p-1/p)$, $POT_i(p-1/p)$

Table 3 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)

Table 4 Task sets consisting of real benchmarks

Sets	Tasks
Set 1	cjpeg, epic, pegwit, rawcaudio, mpeg2, toast
Set 2	CRC32, dijkstra, FFT, pktflow, qsort, rijndael
Set 3	A2TIME01, AIFFTR01, AIFIRF01, BITMNP01, IDCTRN01, RSPEED01
Set 4	cjpeg, pegwit, qsort, susan, A2TIME01, IDCTRN01
Set 5	epic, rawcaudio, dijkstra, CRC32, AIFFTR01, BITMNP01
Set 6	mpeg2, toast, pktflow, rijndael, AIFIRF01, RSPEED01
Set 7	pegwit, mpeg2, qsort, FFT, BITMNP01, IDCTRN01

Table 5 Cache hierarchy configuration explored using different exploration methods.

	Exhaust	SLOT	TST	ILOT	ILT
cjpeg	4752	288	192	54	31
epic	4752	288	70	54	31
pegwit	4752	288	128	36	36
rawcaudi	4752	288	452	54	33
CRC32	4752	288	318	54	33
dijkstra	4752	288	92	54	32
FFT	4752	288	165	52	36
pktflow	4752	288	114	54	37
qsort	4752	288	116	54	37
rijndael	4752	288	58	54	31
susan	4752	288	352	54	33
A2TIME	4752	288	92	54	34
AIFFTR	4752	288	120	54	31
AIFIRF0	4752	288	79	54	38
BITMNP	4752	288	68	54	38
IDCTRN	4752	288	84	54	36
RSPEED	4752	288	116	53	37

ALGORITHMS

Algorithm 1: Cache configuration selection

Input: Task list entry, ready task list and profile table.
Output: An appropriate cache configuration combination.
Step 1: Select executing task T_c.
if The algorithm is called when preemption happens **then**
    Calculate the preempted task T_p's CP.
    for i = 0 to p – 1 **do**
        if $TIN_{T_p} \times i / p \leq EIN_{T_p} < TIN_{T_p} \times (i + 1) / p$ **then**
            $CP_{T_p} = i$ / p;
        end if
    end for
    T_c = preempting task;
else
    T_c = the task with maximum priority from RTL;
end if
Step 2: Sort all task 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 $t + POT_{T_c}(CP_{T_c}/p) + \sum_{i=1}^{j} POT_{T_i}((CP_{T_i} + 1)/p) > D_{T_j}$ **then**
        Task T_j is subject to be discarded;
    end if
end for
Step 3: Select cache configuration for T_c. Let m' be the number of tasks in RTL left after **Step 2**.
if $t + EOT_{T_c}(CP_{T_c}/p) > D_{T_c}$ **then**
    EP_OK = *false*;
else
    EP_OK = *true*;
    for j = 1 to m' **do**
        if $t + EOT_{T_c}(CP_{T_c}/p) + \sum_{i=1}^{j} POT_{T_i}((CP_{T_i} + 1)/p) > D_{T_j}$ **then**
            EO_OK = *false*;
        end if
    end for
end if
if *EO_OK* **then**
    $Cache_{T_c}^{il1} = EO_{T_c}^{il1}(CP_{T_c}/p)$; $Cache_{T_c}^{dl1} = EO_{T_c}^{dl1}(CP_{T_c}/p)$; $Cache_{T_c}^{ul2} = EO_{T_c}^{ul2}(CP_{T_c}/p)$;
else
    $Cache_{T_c}^{il1} = PO_{T_c}^{il1}(CP_{T_c}/p)$; $Cache_{T_c}^{dl1} = PO_{T_c}^{dl1}(CP_{T_c}/p)$; $Cache_{T_c}^{ul2} = PO_{T_c}^{ul2}(CP_{T_c}/p)$;
end if
Return: $Cache_{T_c}^{il1}$, $Cache_{T_c}^{dl1}$, $Cache_{T_c}^{ul2}$

BIOGRAPHIES

Weixun Wang received his B.E. degree in software engineering from the Software Institute, Nanjing University, Nanjing, China, in 2007. He is currently pursuing his Ph.D. degree in the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, USA. His research interests include the area of design automation of embedded systems with focus on dynamic cache reconfiguration, energy optimization, temperature management, design space exploration and lossless data compression.

Prabhat Mishra is an Associate Professor in the Department of Computer and Information Science and Engineering at the University of Florida. His research interests include design automation of embedded systems, hardware/software verification, and low-power reconfigurable architectures. He received his B.E. from Jadavpur University, Kolkata in 1994, M.Tech. from the Indian Institute of Technology, Kharagpur in 1996, and Ph.D. from the University of California, Irvine in 2004 -- all in computer science and engineering. Prior to joining University of Florida, he spent several years in various semiconductor and design automation companies including Intel, Motorola, Synopsys and Texas Instruments. He has published two books, nine book chapters and more than 60 research articles in premier international journals and conferences. His research has been recognized by several awards including the 2003 CODES+ISSS Best Paper Award, 2005 European Design Automation Association Outstanding Dissertation Award, and 2008 National Science Foundation CAREER Award. He has also received the International Educator of the Year Award from the UF College of Engineering for his significant international research and teaching contributions.

Prof. Mishra currently serves as the Information Director of ACM Transactions on Design Automation of Electronic Systems, Guest Editor of IEEE Design & Test of Computers, and as a program/organizing committee member of several premier ACM and IEEE conferences including

DATE, ASPDAC, CODES+ISSS, VLSI Design, VLSI-SoC, GLSVLSI, and ISVLSI. He has also served as General Chair of IEEE High Level Design Validation and Test (HLDVT) 2010, Program Chair of HLDVT 2009, and Guest Editor of Springer Journal of Electronic Testing and International Journal of Parallel Programming. He is a senior member of ACM, and a senior member of IEEE.