Dynamic Cache Reconfiguration and Partitioning for Energy Optimization in Real-Time Multicore Systems*

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

ABSTRACT

Multicore architectures, especially chip multi-processors, have been widely acknowledged as a successful design paradigm. Existing approaches primarily target application-driven partitioning of the shared cache to alleviate inter-core cache interference so that both performance and energy efficiency are improved. Dynamic cache reconfiguration is a promising technique in reducing energy consumption of the cache subsystem for uniprocessor systems. In this paper, we present a novel energy optimization technique which employs both dynamic reconfiguration of private caches and partitioning of the shared cache for multicore systems with real-time tasks. Our static profiling based algorithm is designed to judiciously find beneficial cache configurations (of private caches) for each task as well as partition factors (of the shared cache) for each core so that the energy consumption is minimized while task deadline is satisfied. Experimental results using real benchmarks demonstrate that our approach can achieve 29.29% energy saving on average compared to systems employing only cache partitioning.

Categories and Subject Descriptors

C.3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time systems and embedded systems

General Terms

Algorithm, Design

Keywords

Multicore systems, real-time systems, energy optimization, cache, dynamic reconfiguration

1. INTRODUCTION

Due to various limitations in device scaling faced by semiconductor microelectronic design nowadays, computation using single-core processors has hit the wall on its way of performance improvement. Chip multiprocessor (CMP) architectures, which integrates multiple processing units on a single chip, have been widely adopted by major vendors like Intel, AMD, IBM and ARM in both general-purpose computers [1] as well as embedded systems [2] [3]. Multicore processors are able to run multiple threads in parallel at lower power dissipation per unit of performance. Despite the advantages, energy conservation is still a primary concern in system optimization. While being important for conventional computers, energy efficiency is especially critical for embedded systems since many of them are driven by batteries. Real-time systems which runs applications with timing constraints require unique considerations since deadlines have to be satisfied to avoid catastrophic consequences and ensure correct system behavior. Due to the ever growing demands for parallel computing, multicore processors are commonly employed in real-time systems [4].

The ever increasing gap between processor and memory speed leads to the prevalence of on-chip caches. Multi-level caches are responsible for a significant part nowadays (over 50%) of the overall system energy consumption due to its large on-chip area and high access frequency [5]. One of the most effective techniques for cache energy optimization is dynamic cache reconfiguration (**DCR**) [6]. By tuning the cache configuration at runtime, we can satisfy memory access behavior of different applications so that significant amount of energy savings can be achieved. DCR has been well studied for uniprocessor in both general-purpose computers [7] as well as real-time embedded systems [8] [9]. Typically, L2 cache acts as a shared resource in CMP. Recent research has showed that shared on-chip cache may become a performance bottleneck for CMP systems because of contentions among parallel running tasks [10] [11]. To alleviate this problem, cache partitioning (**CP**) techniques judiciously partition the shared cache and maps a designated part of the cache to each core. CP is designed at the aim of performance improvement [12], inter-task interference elimination [10], thread-wise fairness optimization [13], off-chip memory bandwidth minimization [14] and energy consumption reduction [15]. Meanwhile, CP is also beneficial for real-time systems to improve worst-case execution time (WCET) analysis, system predictability and cache utilization [16] [15].

In this paper, we present a novel energy optimization technique which efficiently integrates cache partitioning and dynamic reconfiguration in real-time multicore systems. To the best of our knowledge, this is the first work that employs DCR and CP simultaneously. Our contribution can be summarized as:

1. We find that DCR in L1 caches has great impact on decisions of CP in shared L2 and vice versa. Moreover, both DCR and CP play important roles in energy conservation.
2. Our approach can minimize the cache hierarchy energy consumption while guarantee all timing constraints.
3. We propose efficient static profiling techniques and algorithms to find beneficial L1 cache configurations for each task and L2 partition factors for each core.
4. Our approach considers multiple tasks on each core thus is more general than existing CP techniques which assume only one application per core [11] [15] [14].
5. We study the effect of different deadline constraints.

The remaining part of the paper is organized as follows. Related works are discussed in Section 2. Section 3 describes the architecture model and motivation of our work. Section 4 presents our

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

DAC 2011, June 5-10, 2011, San Diego, California, USA.

approach for CMPs in detail, followed by experimental results in Section 5. Finally, Section 6 concludes the paper.

2. RELATED WORK

Several reconfigurable cache architectures are proposed in recent years [5] [6]. DCR techniques for general-purpose systems based on both dynamic and static analysis are presented in [7] and [17], respectively. In real-time systems, the major challenge of employing DCR is to determine when and how to reconfigure the cache so that energy consumption is minimized while no timing constraint is violated. Wang et al. proposed profile-based scheduling-aware DCR for soft real-time systems with single-level cache [8] as well as multi-level cache hierarchy [9]. Recent research works have also applied DCR in hard real-time systems in conjunction with processor voltage scaling [18]. However, none of them considers multicore platforms.

Cache partitioning techniques are widely studied for various design objectives for multicore processors. Majority of them focused on reducing cache miss rate thus improving performance. Suh et al. [19] utilized hardware counters to gather runtime information which is used to partition the shared cache through the replacement unit. Qureshi et al. [12] proposed a low-overhead CP technique based on online monitoring and cache utilization of each application. Kim et al. [13] focused on fair cache sharing using both dynamic and static partitioning. Recently, CP is employed for low-power system designs. Reddy et al. [10] [15] showed that, by eliminating inter-task cache interferences, both dynamic and leakage energy can be saved. Cache bank structure aware CP in CMP platforms is studied in [11]. Yu et al. [14] targeted at minimizing off-chip bandwidth through off-line profiling. Nevertheless, existing CP techniques only focus on shared L2 cache and ignore the impact as well as the optimization opportunities from reconfiguring private L1 caches.

3. BACKGROUND AND MOTIVATION

In this section, we show important features of the underlying architecture. Next, we describe the cache energy model. We also present an illustrative example to motivate the need and usefulness of our approach.

3.1 Architecture Model

Figure 1 illustrates a typical CMP platform with private L1 caches (IL1 and DL1) in each core and a shared on-chip L2 cache. Here, both instruction L1 and data L1 cache associated with each core are highly reconfigurable in terms of effective capacity, line size and associativity. The underlying reconfigurable cache architecture we adopt is based on [6]. Specifically, gated-V_{dd} technique is used to shut down the banks for cache size tuning. Associativity can be changed by logically concatenating neighboring ways. Line size is reconfigured by specifying the number of unit-length blocks fetched in each cache access. This architecture requires very simple hardware augmentation and minor overhead [6].

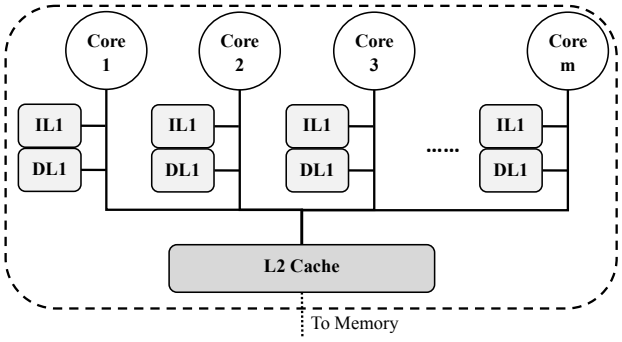

Figure 1: Typical multicore architecture with shared L2 cache.

Unlike traditional LRU replacement policy which implicitly partitions each cache set on a demand basis, we use a way-based partitioning in the shared cache [20]. As shown in Figure 2, each L2 cache set (here with a 8-way associativity) is partitioned in the granularity of ways. Each core is assigned a group of ways and will only access that portion in all cache sets. LRU replacement is enforced in each individual group which is achieved by maintaining separate sets of "age bit". It is also possible to divide the cache by sets (set-based partitioning) in which each set retains full associativity [14]. However, since real-time embedded systems usually have small number of cores, way-based partitioning is beneficial enough for exploiting energy efficiency. We refer the number of ways assigned to each core as its *partition factor*. For example, the L2 partition factor for Core 1 in Figure 2 is 3.

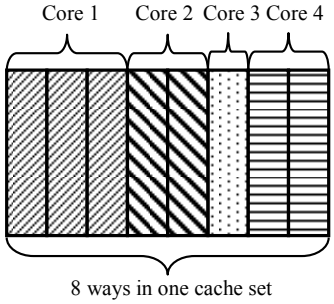

Figure 2: Way-based cache partitioning example (four cores with a 8-way associative shared cache).

In this work, we use static cache partitioning. In other words, L2 partition factors for each core are pre-determined during design time and remain the same through out the system execution. Dynamic partitioning [12] requires online monitoring, runtime analysis and sophisticated OS support thus is not feasible for embedded systems. Furthermore, real-time systems normally have highly deterministic characteristics (e.g., task release time, deadline, input set) which make off-line analysis most suitable [21]. By static profiling, we can potentially search much larger design space and thus achieve better optimization results.

3.2 Energy Model

Cache energy consumption is modeled as the sum of dynamic and static energy: $E = E_{dyn} + E_{sta}$. Dynamic energy dissipation E_{dyn} comes from both cache accesses and cache misses:

$$E_{dyn} = n_{access} \cdot E_{access} + n_{misses} \cdot E_{miss} \quad (1)$$

where n_{access} and n_{miss} are number of cache accesses and misses, respectively. Cache access energy E_{access} is known to be constant for one specific configuration. E_{miss} is computed as:

$$E_{miss} = E_{memside} + E_{block\_fill} + E_{\mu P\_stall} \quad (2)$$

where $E_{memside}$ is the energy required for accessing lower levels of the memory hierarchy, E_{block_fill} is the energy for filling the cache block by the fetched data, and $E_{\mu P\_stall}$ is the energy consumed by the stalled processor during cache miss. For multi-level cache subsystem, as considered in this paper, overlapped energy consumption has to be counted only once (e.g., $E_{memside}$ of L1 cache is essentially part of L2's E_{dyn}). Let P_{sta} denote the static power consumption of cache, E_{sta} is simply computed as $E_{sta} = P_{sta} \cdot t$ where t is the total execution time. Values of E_{access}, P_{sta} and E_{block_fill} are collected from CACTI [22], using 70*nm* technology, for all cache configurations.

3.3 Motivation

Figure 3 shows the number of L2 cache misses and instruction per cycle (IPC) for two benchmarks (*qsort* from MiBench [23] and *vpr* from SPEC CPU2000 [24]) under different L2 cache partition

factors p (in a 8-way associative L2 cache) and randomly chosen four L1 cache configurations[1]. Unallocated L2 cache ways remain idle. We observe that changing L1 cache configuration will lead to different number of L2 cache misses. It is expected because L1 cache configuration determines the number of L2 accesses. System performance (i.e., IPC) is also largely affected by L1 configurations. Notice that for larger partition factors (e.g., 7), the difference in L2 misses is negligible but IPC shows great diversity. It is because not only L2 partitioning but also L1 configurations determine the performance.

It is also interesting to see that *vpr* shows larger variances at the same L2 partition factor than *qsort*. For example, the number of L2 cache misses becomes almost identical (although there are times of differences in the numbers of L2 accesses) at $p = 4$ for *qsort* while it starts to converge for *vpr* only after $p = 6$. The reason behind this is that, for *qsort*, there are mostly compulsory misses for $p \geqslant 4$. In other words, $p = 4$ is a sufficient L2 partition factor for *qsort* in terms of performance. However, increasing p and reducing number of accesses continue to bring benefit for *vpr* as shown in Figure 3(c) due to the fact that *vpr* has more capacity and conflict misses.

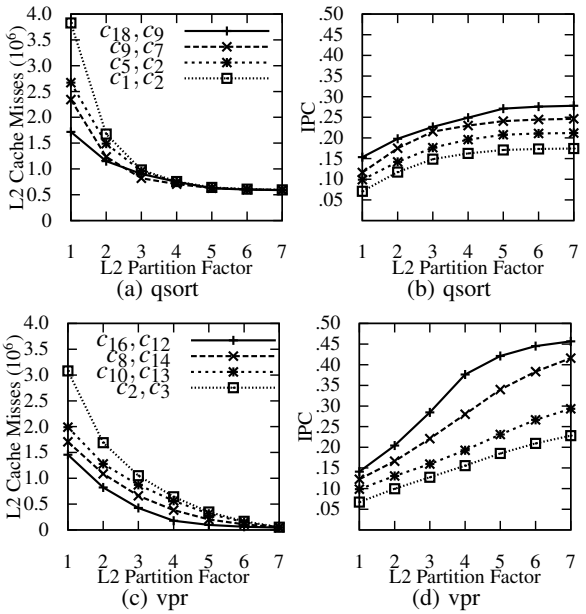

Figure 3: L1 DCR impact on L2 CP in performance.

Given the above observations, we see that L1 DCR has major impact on L2 CP and there are interesting trade-offs that can be explored for optimizations. Therefore, both DCR and CP should be exploited simultaneously for energy conservation in real-time multicore systems. Our experimental results in Section 5.2 confirms our conjecture.

4. DYNAMIC CACHE RECONFIGURATION AND PARTITIONING

In this section, we formulate our energy optimization problem based on performing dynamic reconfiguration of private L1 caches and static partitioning of shared L2 cache (DCR + CP). We present our static profiling strategy. The actual dynamic programming based algorithm which utilizes the static profiling information is then described in detail.

4.1 Problem Formulation

The system we consider can be modeled as:

- A multicore processor with m cores $\mathcal{P}\{p_1, p_2, ..., p_m\}$.
- Each core of the processor has highly-reconfigurable IL1 and DL1 caches both of which supports h different configurations $\mathcal{C}\{c_1, c_2, ..., c_h\}$.
- A α-way associative shared L2 cache with way-based partitioning enabled.
- A set of n independent tasks $\mathcal{T}\{\tau_1, \tau_2, ..., \tau_n\}$ with a common deadline D[2].

Suppose we are given:

- A task mapping $\boldsymbol{M} : \mathcal{T} \rightarrow \mathcal{P}$ in which tasks are mapped to each core. Let ρ_k denotes the number or tasks on core p_k.
- A L1 cache configuration assignment $\boldsymbol{R} : \mathcal{C}_I, \mathcal{C}_D \rightarrow \mathcal{T}$ in which one IL1 and one DL1 configuration are assigned to each task.
- A L2 cache partitioning scheme $\boldsymbol{P}\{f_1, f_2, ..., f_m\}$ in which core $p_i \in \mathcal{P}$ is allocated f_i ways.
- Task $\tau_{k,i} \in \mathcal{T}$ (i^{th} task on core p_k) has execution time of $t_{k,i}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})$. Let $E_{L1}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})$ and $E_{L2}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})$ denote the total energy consumption of all the L1 caches and the shared L2 cache, respectively.

Our goal is to find $\boldsymbol{M}$, $\boldsymbol{R}$ and $\boldsymbol{P}$ such that the overall energy consumption E of the cache subsystem:

$$E = E_{L1}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}) + E_{L2}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P}) \tag{3}$$

is minimized subject to:

$$\max(\sum_{i=1}^{\rho_k} t_{k,i}(\boldsymbol{M}, \boldsymbol{R}, \boldsymbol{P})) \leqslant D \ , \ \forall k \in [1, m] \tag{4}$$

$$\sum_{i=1}^{m} f_i = \alpha \ ; \ f_i \geqslant 1 \ , \ \forall i \in [1, m] \tag{5}$$

Equation (4) guarantees that all the tasks in $\mathcal{T}$ are finished by the deadline D. Equation (5) ensures that the L2 partitioning $\boldsymbol{P}$ is valid.

4.2 Static Profiling

In this paper, we assume that the task mapping $\boldsymbol{M}$ is given. A reasonable (as well as beneficial) task mapping would be a bin packing of the tasks using their base case execution time to all the m cores so that the total execution time in each core is similar. Here the base case execution time refers to the time one task takes in the system where L1 cache reconfiguration is not applied (using the base configuration) and L2 cache is evenly partitioned. Theoretically, we can simply profile the entire task set $\mathcal{T}$ under all possible combinations of $\boldsymbol{R}$ and $\boldsymbol{P}$. Unfortunately, this exhaustive exploration is not feasible due to its excessive simulation time requirement. As an example, in this work, the reconfigurable L1 cache contains four banks each of which is 1 KB. Therefore, it offers effective capacities of 1 KB, 2KB and 4 KB. Line size can be tuned to 16, 32 and 64 bytes while each cache set supports 1-way, 2-way and 4-way associativity. There are totally $h = 18$ different configurations[3]. Even if we have four cores with only two tasks each core and a 8-way associative L2 cache, the total number of multicore architectural simulations will be $((18^2)^2)^4 \times 35$. To be specific, 18^2 denotes the number of IL1 and DL1 cache configurations for each task. $(18^2)^2$ presents all possible L1 cache combinations of the two tasks in each core. Upon that, $(18^2)^2)^4$ denotes all the possible combinations across four cores. According to Equation (5), the size of $\boldsymbol{P}$ (i.e., $|\boldsymbol{P}|$) equals 35 when $m = 4$ and $\alpha = 8$. Obviously, this simulation time is even longer than the age of the universe if each simulation takes only 1 minute.

This problem can be greatly relieved by exploiting the independence of the design space's each dimension. We observe that each task can actually be profiled individually. It is because the tasks that we consider do not have application-specific interactions (e.g.,

[1]Here c_{18} and c_9, for example, stands for the 18^{th} and 9^{th} configuration for IL1 and DL1, respectively.

[2]Our approach can be easily extended for individual deadlines.

[3]It is not equal to 3^3 since not all combinations are valid [9].

data sharing) except the contention for the shared resources. Essentially, using L2 cache partitioning, each core p_i can be viewed as a uniprocessor with f_i-way associative L2 cache (i.e., the capacity is f_i/α of the original). L1 cache activities are private at each core while L2 activities happen independently in each core's partition. Therefore, we simulate each task in $\mathcal{T}$ independently under all combinations of L1 cache configurations and L2 cache partition factors (from 1 to $\alpha-1$). In other words, the total number of single-core simulations equals to $h^2 \cdot (\alpha-1) \cdot n$. Using the same example above, with 8 tasks, it is $(18^2) \times 7 \times 8$. Note that this profiling process is independent of the task mapping $\boldsymbol{M}$ and the number of cores m. Apparently, it will take only reasonable static profiling time (e.g., at most three days).

4.3 DCR + CP Algorithm

Static profiling results are used to generate profile tables for each task. Each entry in the profile table records the cache energy consumption, for both L1 and the L2 partition, as well as the corresponding execution time of that task. The dynamic energy of L2 cache is computed using Equation (1) based on the statistics (accesses) from the core to which the task is assigned. The static energy, however, is estimated by treating the allocated ways as a standalone cache. In other words, the static energy is proportional to the number of allocated ways. There are $h^2 \cdot (\alpha-1)$ entries in every task's profile table. For task $\tau_{k,i} \in \mathcal{T}$ (i^{th} task on core p_k), let $e_{k,i}(h_1,h_2,f_k)$ denote the total cache energy consumption if task $\tau_{k,i}$ is executed with (IL1,DL1) configurations (c_{h_1}, c_{h_2}) and L2 partition factor f_k. Similarly, let $t_{k,j}(h_1,h_2,f_k)$ denote the execution time. Our problem now can be presented as to minimize:

$$E = \sum_{k=1}^{m} \sum_{i=1}^{\rho_k} e_{k,i}(h_1,h_2,f_k) \tag{6}$$

subject to:

$$\max(\sum_{i=1}^{\rho_k} t_{k,i}(h_1,h_2,f_k)) \leqslant D \ , \ \forall k \in [1,m] \tag{7}$$

$$\sum_{i=1}^{m} f_i = \alpha \ ; \ f_i \geqslant 1 \ , \ \forall i \in [1,m] \tag{8}$$

Our algorithm consists of two steps. Since static partitioning is used, all the tasks on each core share the same L2 partition factor f_k. This fact gives us an opportunity to simplify our algorithm without losing any precision. In the **first step**, we find the optimal L1 cache assignments for the tasks on each core separately under all L2 partition factors. Specifically, we find $\boldsymbol{R}$ to minimize $E_k(f_k) = \sum_{i=1}^{\rho_k} e_{k,i}(c_1,c_2,f_k)$ constrained by $\sum_{i=1}^{\rho_k} t_{k,i}(c_1,c_2,f_k) \leqslant D$ with k and f_k fixed for $\forall p_k \in \mathcal{P}$ and $\forall f_k \in [1,\alpha-1]$. This step (sub-problem) is illustrated in Figure 4 for p_m with $f_m = 2$. Similar to the uniprocessor DVS problem [25], each instance of this subproblem can be reduced from the multiple-choice knapsack problem (MCKP) and thus is NP-hard.

Since the subproblem size (measured by h^2, ρ_k) and the embedded application size (measured by energy value) are typically small, a dynamic programming algorithm can find the optimal solution quite efficiently as follows. Let $e_k^{max}(f_k)$ and $e_k^{min}(f_k)$ be defined as $\sum_{i=1}^{\rho_k} max\{e_{k,i}(h_1,h_2,f_k)\}$ and $\sum_{i=1}^{\rho_k} min\{e_{k,i}(h_1,h_2,f_k)\}$, respectively. Hence, $E_k(f_k)$ is bounded by $[e_k^{min}(f_k), e_k^{max}(f_k)]$. In order to guarantee the timing constraint, the energy value is discretized in our dynamic programming algorithm. Let $S_j^{E_k}$ denote the partial solution for the first j tasks which has an accumulative energy consumption equal to E_k while the execution time is minimized. We create a two-dimensional table T in which each element $T[j][E_k]$ stores the execution time of $S_j^{E_k}$. The recursive relation for dynamic programming thus is:

$$T[j][E_k] = \min_{h_1,h_2 \in [1,h]} \{T[j-1][E_k - e_{k,i}(h_1,h_2,f_k)] + t_{k,i}(h_1,h_2,f_k)\} \tag{9}$$

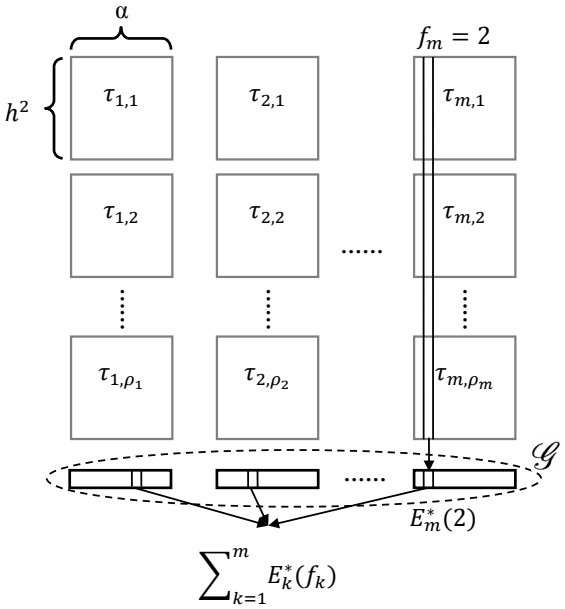

Figure 4: Illustration of our algorithm.

Initially, all entries in T store some value larger than D. Based on the above recursion, we fill up the table $T[j][E_k]$ in a row by row manner for all energy values in $[e_k^{min}(f_k), e_k^{max}(f_k)]$. During the process, all previous $i-1$ rows are filled when the i^{th} row is being calculated. Finally, the optimal energy consumption $E_k^*(f_k)$ is found by:

$$E_k^*(f_k) = \{\min E_k \mid T[\rho_k][E_k] \leqslant D\} \tag{10}$$

Algorithm 1 outlines the major steps in our DCR + CP approach. Our algorithm iterates over all tasks in core p_k (1 to ρ_k). During each iteration, all discretized E_k values and L1 cache configurations (1 to h^2) for current task are examined. Therefore, the time complexity is $O(\rho_k \cdot h^2 \cdot (e_k^{max}(f_k) - e_k^{min}(f_k)))$. Note that energy values (reflected in the last term of the complexity) can always be measured in certain unit so that they are numerically small to make the dynamic programming efficient. The size of table T decides the memory requirement, which is $\rho_k \cdot (e_k^{max}(f_k) - e_k^{min}(f_k)) \cdot sizeof(element)$ bytes. In each entry of T, we can use minimum number of bits to remember the L1 configuration index instead of real execution time values. For calculation purposes, two two-dimensional arrays are used for temporarily storing time values for current and previous iterations. The above process is repeated for $\forall k \in [1,m]$ and $\forall f_k \in [1,\alpha-1]$. It is possible that, for some f_k, there is no feasible solution for core p_k satisfying the deadline. We mark them as invalid. The results form a new profile table $\mathcal{G}$ for each core in which there are $[1,\alpha-1]$ entries and each entry stores the corresponding optimal solution $E_k^*(f_k)$, as shown in Figure 4.

In the **second step**, the global optimal solution E^* can be found by calculating the overall energy consumption for all L2 partition schemes in $\boldsymbol{P}$ which complies with Equation (8). Given a partition factor f_k for core p_k, the optimal energy consumption $E_k^*(f_k)$ observing D has been calculated in the first step. Invalid partition schemes are discarded. We have $E^* = \min\{\sum_{k=1}^{m} E_k^*(f_k)\}$ for $\{f_1,f_2,...,f_m\} \in \boldsymbol{P}$. Therefore, for each L2 partitioning scheme, the corresponding solution can be found in $O(m)$ time. Since the size of $\boldsymbol{P}$ is small (e.g., 455 and 4495 for 4 cores with 16-way and 32-way associative L2 cache, respectively), an exhaustive exploration is efficient enough for this step to find the minimum cache hierarchy energy consumption in $O(m \cdot |\boldsymbol{P}|)$ time. Otherwise, a dynamic programming algorithm can be used. Note that E^* is not strictly equal to the actual energy dissipation since the L2 cache still consumes static power in its entirety after some cores finish their tasks. Therefore, in our experimental results, we have added this portion

of static energy to make it accurate. If L2 cache lines are turned off in those partitions using techniques such as cache decay [26] to save static power dissipation, E^* is already accurate. Each core along with its private caches are assumed to be turned off after it finishes execution.

Algorithm 1 DCR + CP Algorithm.

for $k = 1$ to m **do**
  for $f_k = 1$ to $\alpha - 1$ **do**
    for $l = e_k^{min}(f_k)$ to $e_k^{max}(f_k)$ **do**
      for $h_1, h_2 \in [1, h]$ **do**
        if $e_{k,1}(h_1, h_2, f_k) == l$ **then**
          if $t_{k,1}(h_1, h_2, f_k) < T[1][l]$ **then**
            $T[1][l] = t_{k,1}(h_1, h_2, f_k)$
          end if
        end if
      end for
    end for
    for $i = 2$ to ρ_k **do**
      for $l = e_k^{min}(f_k)$ to $e_k^{max}(f_k)$ **do**
        for $h_1, h_2 \in [1, h]$ **do**
          $last = l - e_{k,i}(h_1, h_2, f_k)$
          if $T[j-1][last] + t_{k,i}(h_1, h_2, f_k) < T[j][l]$ **then**
            $T[j][l] = T[j-1][last] + t_{k,i}(h_1, h_2, f_k)$
          end if
        end for
      end for
    end for
    $E_k^*(f_k) = \min\{E_k \mid T[\rho_k][E_k] \leqslant D\}$
  end for
end for
for all $\boldsymbol{P}_i\{f_1, f_2, ..., f_m\} \in \boldsymbol{P}$ **do**
  $E_i^* = \sum_{k=1}^{m} E_k^*(f_k)$
end for
return $\min\{E_i^*\}$

5. EXPERIMENTS

5.1 Experimental Setup

To evaluate our approach's effectiveness, we use 20 benchmarks selected from MiBench [23] – *basicmath*, *bitcount*, *CRC32*, *dijkstra*, *FFT*, *patricia*, *qsort*, *sha*, *stringsearch*, *toast* and *untoast* – and SPEC CPU 2000 [24] – *ammp*, *applu*, *gcc*, *lucas*, *mcf*, *parser*, *swim*, *vpr* and *mgrid*. In order to make the size of SPEC benchmarks comparable with MiBench, we use reduced (but well verified) input sets from MinneSPEC [27]. Table 1 lists the task sets used in our experiments which are combinations of the selected benchmarks. We choose 4 task sets where each core contains 2 benchmarks, 3 task sets where each core contains 3 benchmarks and 2 task sets where each core contains 4 benchmarks. As mentioned in Section 4.2, the task mapping is based on the rule that the total execution time of each core is comparable. The deadline D is set in a way that there is a feasible L1 cache assignment for every partition factor in every core. In other words, all possible L2 partition schemes can be used. We will examine the effect of timing constraints (deadlines) in Section 5.3.

M5 [28], a widely used architectural simulator, is adopted in our experiments. We enhanced M5 to make it support shared cache partitioning and different line sizes in different caches (IL1, DL1 and L2) to support cache reconfiguration in CMP mode. We configure the simulated system with a four-core processor each of which runs at 500MHz. The TimingSimpleCPU model [28] in M5 is used which represents an in-order core which stalls during cache accesses and memory response handling. The L2 cache configuration is assumed to be 32KB, 8-way associative with 32-byte lines. The memory size is set to 256MB. The L1 cache, L2 cache and memory access latency are set to 2ns, 20ns and 200ns, respectively.

5.2 Energy Savings

We compare the following three approaches.

- **CP**: L2 cache partitioning only (optimal partition).
- **DCR + UCP**: L1 DCR with uniform L2 cache partitioning.
- **DCR + CP**: Our approach.

Here CP only approach uses optimal L2 partition scheme with all L1s in base configuration. It can be achieved using our algorithm in Section 4.3 without the first step. Figure 5 illustrates this comparison in energy consumption for all task sets in Table 1. Energy values are normalized to CP. As discussed in Section 4.2, our reconfigurable L1 cache has a base size of 4KB. Here, we examine two kinds of L1 base configurations: 4KB with 2-way associativity and 32-byte line size (4KB_2W_32B), and 4KB with 4-way associativity and 64-byte line size (4KB_4W_64B). In the former case, DCR + CP can save 18.35% of cache energy on average compared with CP. In the latter case, up to 33.51% energy saving (e.g., for task set 4) can be achieved and averagely 29.29%. Compared with DCR + UCP, our approach is able to achieve up to 14% more energy savings by carefully select cache partitioning scheme $\boldsymbol{P}$. Note that although results for only two L1 base configurations are shown here, we observe that similar improvements can be achieved for other base configurations (e.g., 19.30% for 2KB_2W_32B).

It is valuable to disclose the energy reduction ability of our approach. Using task set 4 in Figure 5 (b) as an example, CP selects the best $\boldsymbol{P} = \{1, 5, 1, 1\}$ with L1 configuration of 4KB_4W_64B. It consumes total energy of 125.8 *mJ* and finishes all tasks in 1600 *ms*. With DCR + CP, the optimal $\boldsymbol{P} = \{2, 4, 1, 1\}$ and the L1 caches are configured differently for each task. For example, *FFT* on Core 1 uses 1KB_1W_64B and 4KB_4W_16B of IL1 and DL1, respectively, while *swim* on Core 3 uses 4KB_4W_16B and 2KB_2W_32B. Using DCR + CP, the energy requirement is reduced to 83.6 *mJ* and all tasks finishes in 1788 *ms*.

5.3 Deadline Effect

It is also meaningful to see how deadline constraint can affect the effectiveness of our approach. Using the same example above, for task set 4, we vary the deadline from 1800 *ms* to 1520 *ms* in step of 10 *ms* (there is no solution for deadlines shorter than 1520 *ms*). Figure 6 shows the result for both CP and DCR + CP. We can observe that our approach can find efficient solutions and outperforms CP consistently at all deadline levels.

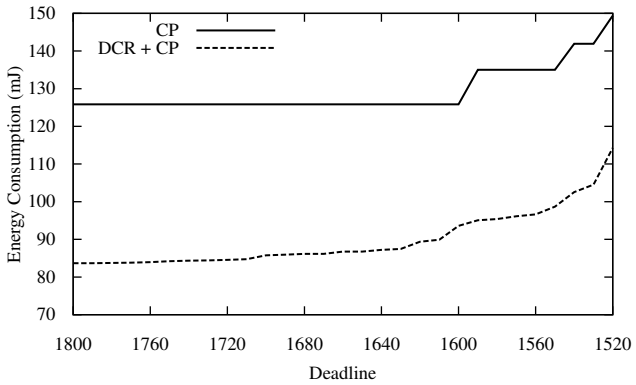

Figure 6: Deadline effect on total energy consumption.

6. CONCLUSION

In this paper, we presented an efficient approach to integrate dynamic cache reconfiguration (DCR) and cache partitioning (CP) for real-time multicore systems. We discovered that there is a strong correlation between L1 DCR and L2 CP. While CP is effective in reducing inter-task interferences, applying DCR together can further improve the energy efficiency without violating timing constraints. Our static profiling technique drastically reduces the ex-

Table 1: Multi-task benchmark sets.

	Core 1	Core 2	Core 3	Core 4
Set 1	qsort, vpr	parser, toast	untoast, swim	dijkstra, sha
Set 2	mcf, sha	gcc, bitcount	patricia, lucas	basicmath, swim
Set 3	applu, lucas	dijkstra, swim	ammp, FFT	basicmath, stringsearch
Set 4	mgrid, FFT	dijkstra, parser	CRC32, swim	applu, bitcount
Set 5	mcf, toast, sha	gcc, parser, stringsearch	patricia, qsort, vpr	basicmath, CRC32, ammp
Set 6	mgrid, parser, gcc	toast, FFT, mcf	bitcount, ammp, patricia	applu, dijkstra, qsort
Set 7	vpr, sha, untoast	CRC32, lucas, qsort	mgrid, bitcount, FFT	applu, parser, stringsearch
Set 8	sha, mcf, untoast, basicmath	toast, gcc, bitcount, patricia	lucas, FFT, CRC32, ammp	vpr, applu, mgrid, swim
Set 9	gcc, stringsearch, parser, dijkstra	untoast, mcf, ammp, bitcount	lucas, patricia, qsort, vpr	basicmath, toast, applu, CRC32

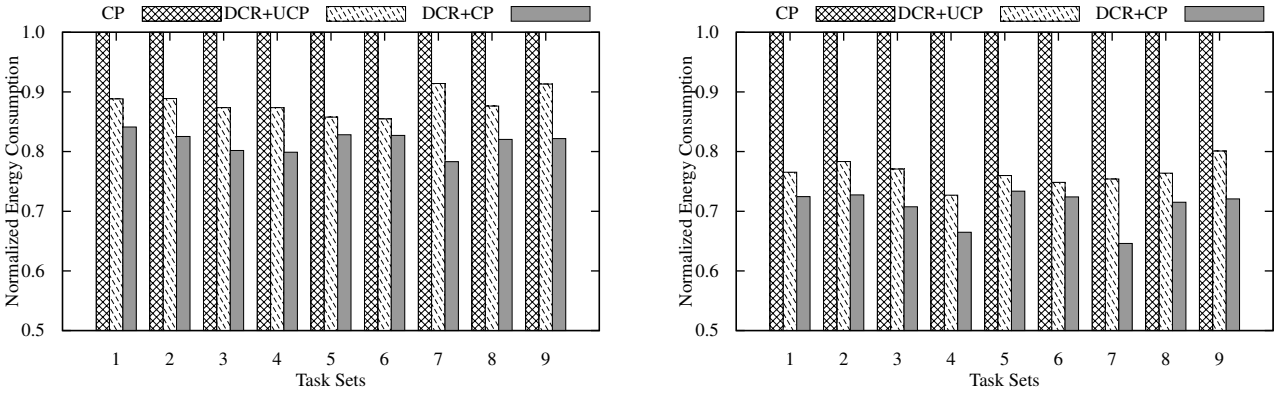

Figure 5: Cache hierarchy energy reduction with L1 base configuration of: (a) 4KB_2W_32B; (b) 4KB_4W_64B.

ploration space without losing any precision. Our DCR + CP algorithm, which can find the optimal L1 configurations for each task and L2 partition factors for each core, is based on dynamic programming with discretization in the energy values. We also studied the effect of deadline variation. Extensive experimental results demonstrated that our approach can achieve 18.35% - 29.29% average energy savings compared with traditional cache partitioning techniques.

7. REFERENCES

[1] Intel. Intel Core i7 processor. www.intel.com.

[2] ARM. ARM11MPCore processor. http://www.arm.com/.

[3] MIPS. MIPS32 1004K. http://www.mips.com/.

[4] Y.-H. Wei et al., "Energy-efficient real-time scheduling of multimedia tasks on multi-core processors," *SAC*, 2010.

[5] A. Malik et al., "A low power unified cache architecture providing power and performance flexibility," *ISLPED*, 2000.

[6] C. Zhang et al., "A highly configurable cache for low energy embedded systems," *ACM TECS*, vol. 6, pp. 362–387, 2005.

[7] A. Gordon-Ross and F. Vahid, "A self-tuning configurable cache," *DAC*, 2007.

[8] W. Wang et al., "Dynamic cache reconfiguration for soft real-time systems," *ACM TECS*, 2011.

[9] W. Wang and P. Mishra, "Dynamic reconfiguration of two-level cache hierarchy in real-time embedded systems," *J. of Low Power Electron.*, vol. 7, no. 1, 2011.

[10] R. Reddy and P. Petrov, "Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems," *CASES*, 2007.

[11] D. Kaseridis et al., "Bank-aware dynamic cache partitioning for multicore architectures," *ICPP*, 2009.

[12] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," *Micro*, 2006.

[13] S. Kim et al., "Fair cache sharing and partitioning in a chip multiprocessor architecture," *PACT*, 2004.

[14] C. Yu and P. Petrov, "Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms," *DAC*, 2010.

[15] R. Reddy and P. Petrov, "Cache partitioning for energy-efficient and interference-free embedded multitasking," *ACM TECS*, vol. 9, no. 3, pp. 1–35, 2010.

[16] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," *DAC*, 2008.

[17] A. Gordon-Ross et al., "Fast configurable-cache tuning with a unified second-level cache," *ISLPED*, 2005.

[18] W. Wang and P. Mishra, "System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems," *IEEE Trans. on VLSI Syst.*, 2011.

[19] G. Suh et al., "A new memory monitoring scheme for memory-aware scheduling and partitioning," HPCA, 2002.

[20] A. Settle et al., "A dynamically reconfigurable cache for multithreaded processors," *J. of Embed. Comput.*, vol. 2, pp. 221–233, 2006.

[21] G. Quan and X. S. Hu, "Energy efficient dvs schedule for fixed-priority real-time systems," *ACM TODAES*, vol. 6, pp. 1–30, 2007.

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

[23] M. Guthaus et al., "Mibench: A free, commercially representative embedded benchmark suite," *WWC*, 2001.

[24] SPEC. SPEC CPU2000. http://www.spec.org/.

[25] W. Wang and P. Mishra, "PreDVS: Preemptive dynamic voltage scaling for real-time systems using approximation scheme," *DAC*, 2010.

[26] S. Kaxiras et al., "Cache decay: exploiting generational behavior to reduce cache leakage power," *ISCA*, 2001.

[27] A. KleinOsowski and D. Lilja, "Minnespec: A new spec benchmark workload for simulation-based computer architecture research," *IEEE CAL*, vol. 1, no. 1, 2002.

[28] N. Binkert et al., "The M5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52 –60, 2006.