Cache Partitioning and Scheduling for Energy Optimization of Real-Time MPSoCs

Gang Chen
TU Munich, Germany
cheng@in.tum.de

Kai Huang
TU Munich, Germany
huangk@in.tum.de

Jia Huang
fortiss GmbH, Germany
huang@fortiss.org

Alois Knoll
TU Munich, Germany
knoll@in.tum.de

***Abstract*—Cache partitioning is a promising technique to reduce energy consumption of the cache subsystem for MPSoCs. Currently, most existing techniques focus primarily on static partition on core level. In this paper, we present a task-level approach and show that it outperforms core-level strategies. By taking the interference patterns of individual tasks into account, our approach generates optimal task-level cache partition schemes as well as feasible schedules at compilation time by means of a mixed integer linear programming formulation. We also present techniques to prune the exploration space of our formulation. Experimental results using real-world benchmarks demonstrate that our approach achieves 33% energy savings on average compared to core-based cache partition approaches.**

I. Introduction

Multi-core architectures are believed to be one of the major solutions for future embedded systems, due to their advantages in performance and power consumption. Memory subsystem is an essential part of such architectures. To alleviate the high latency of the off-chip memory, multi-core architectures are typically equipped with small L1 caches for every core and a relatively large L2 cache shared among all cores. ARM Cortex-A15 series [3] and openSPARC series [14] are examples of this class of architectures. Although the cache subsystem can significantly improve the performance, its energy consumption is a concern. Several studies [16], [20] show that the energy consumption of the cache subsystem accounts for over 50% of the overall chip, due to its large on-chip area and high access frequency. Therefore, reducing the energy consumption of the cache subsystem, in particular the L2 cache, is critical to further prolong the lifespan of the system.

Another known issue of the cache subsystem is that the cache hierarchy complicates the behavior of a system, resulting in difficulties for the analysis of system properties, e.g., timing and energy consumption. This complication is further aggravated when the L2 cache is shared by multiple cores and suffers the consequent inter-core interferences. For instance, cache prefetches from one core can displace prefetches from another core at runtime. Cache partitioning is a promising technique to tackle the aforementioned problem, which partitions the shared L2 cache into separate regions and designates one or a few regions to individual cores. Cache partitioning also has the advantage that it can provide spatial isolation of the cache, which is required by safety standards such as ARINC 653 in the avionic domain.

This work has been partly funded by German BMBF projects ECU (grant number: 13N11936) and Car2X (grant number: 13N11933).

Most of the state-of-art techniques [7], [11], [16] on cache partitioning consider the problem at core level. However, designating a region with a constant size to individual cores is often ineffective w.r.t the energy consumption, since the tasks assigned on the same core might have different requirement and sensitivity to the amount of cache allocated. We argue that by carefully designing a task schedule and reconfiguring the L2 cache partitioning for each task according to the schedule at runtime, the energy consumption of the system can be significantly reduced, compared to the core-level strategies. A motivation example will further elaborate this issue in Section IV.

This paper tackles the problem of schedule-aware cache partitioning for energy optimization on MPSoCs. We present, to the best of our knowledge, the first work that dynamically partitions the shared L2 cache at the task level to improve energy efficiency, taking into account the task dependencies and time-triggered schedule. For a given set of applications represented as directed acyclic graphics and a mapping of the applications on the MPSoC, our approach can generate a time-triggered non-preemptive schedule and a set of cache configurations in the compilation time. At runtime, the L2 cache is dynamically reconfigured at every task invocation according to the generated configurations. The generated schedule and the cache configurations minimize the energy consumption of the cache subsystem while guaranteeing the timing constraints of the applications. The contributions of our work are summarized as:

- We define an energy-minimization problem that combines the cache partitioning and task scheduling and solve it by means of mixed integer linear programming.
- We present a refinement technique to prune the design space, such that our approach is more scalable with respect to the number of applications and the number of ways of the L2 cache.
- We conduct extensive simulations using real-life applications and demonstrate the effectiveness of our approach by comparison with the previous work.

The rest of the paper is organized as follows: Section II reviews related work in the literature. Section III presents basic models and the definition of studied problem. Section IV presents the motivation example and Section V describes the proposed approach. Experimental evaluation is presented in Section VI and Section VII concludes the paper.

II. Related Work

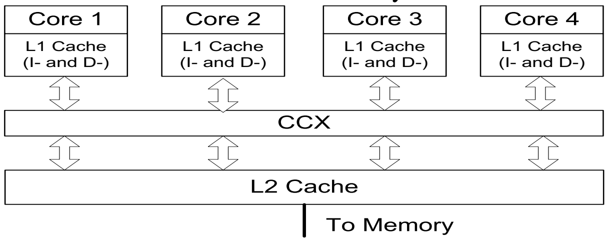
Reconfigurable Cache: Numbers of reconfigurable cache architecture have been proposed in the literature. Zhang et al. [20] propose a highly reconfigurable cache architecture where the cache ways could be tuned via hardware configuration registers. The number of partitioned cache ways is constraint to be a power of two. Albonesi et al. [2] propose a selective cache ways architecture where the cache can be reconfigured to have any number of cache ways. It can be realized with only minor changes in a conventional cache. To improve the energy efficiency, the cache is reconfigured dynamically according to the behavior of a task [2], [20]. Instead of tuning the number of cache ways, the authors in [17] present a selective-sets cache architecture which varies the number of cache sets. Yang et al. [18] propose an hybrid selective-sets-and-ways organization for resizable caches and explore the design alternatives. For the sake of low design complexity and effectiveness, We focus on the cache architecture with selective ways in [2] in this paper.

Cache Partitioning: Cache partitioning techniques are extensively studied to improve the performance of real-time embedded systems. Bui et al. [4] exploit cache partitioning to minimize the task utilization while taking into account the tasks' criticality. By leveraging configurable cache architectures, the authors in [12] propose a technique to eliminate inter-task cache interference and reduce cache energy consumption. Wang et al. [15] propose a profile-based scheduling-aware dynamic cache reconfiguration technique to reduce the cache energy consumption for soft real-time systems. However, none of above works consider the multicore platforms. For the multicore systems, Liu et al. [11] propose a joint task assignment and cache partitioning technique to minimize the overall WCET, where the shared cache is partitioned on core level and cache locking is applied to guarantee a precise WCET. Given a task-level cache partitioning, the authors in [8] develop a sufficient schedulability test for non-preemptive fixed priority scheduling for multicores. However, the work does not consider how to partition the cache to individual tasks. The authors in [7] propose a two-level utilization control solution for energy optimization in multicore real-time systems, in which the cache assigned to a task is upper-bounded by the cache quota of the core. Wang et al. [16] propose an approach to optimize the energy of the cache subsystem for multicore systems. In this work, they dynamically reconfigure the private L1-cache on task-level and statically partition the shared L2-cache to cores. All the works above model tasks as a set of discrete independent tasks, which can not fully exploit the parallel feature of multicore system. Besides, most of them consider cache partitioning on core level.

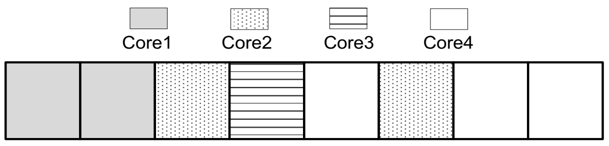
III. Models and Problem Definition

A. Hardware Model

This paper considers a typical multicore system with private L1 cache and a shared on-chip L2 cache, as shown in Fig. 1(a). The multicore system consists of 4 cores $P = \{p_1, p_2, p_3, p_4\}$ and cross switch serves as a high bandwidth interface between the 4 cores and the 4-port L2 cache. This type of interface has been used in the OpenSPARC processors [14] (called CCX). In this work, we focus on partitioning the shared L2 cache and assume the task occupies the entire private L1 cache (both instrument and data cache). Considering the design complexity and effectiveness, we use a *resizable* cache architecture with selective ways as considered in [2] and use a way-based partitioning strategy. As shown in Fig. 1(b), the L2 cache is partitioned in the ways. Each core can dynamically tune the number of selective-ways by reconfiguring *way-mask*. For example, core 2 can select the 3rd and 6th way by setting *way-mask* as $0x24$. In this work, we dynamically assign cache ways to tasks. We show that the existing work that statically partition the L2 cache on core-level [7], [11], [16] is suboptimal. Our approach can also be extended to handle cache architectures that divide cache by sets.



(a) Multicore architecture with 2-level cache



(b) Way-based cache partition

Fig. 1. Hardware model.

B. Task Model

We consider the functionality of the entire system as an *applications set* A, which consists of a set of independent periodic *applications*. An *application* $J \in A$ is modeled as a directed acyclic task graph $G(V, E, H)$, where vertexes V denote the set of tasks T to be executed, the edges E represent data dependencies between tasks and H denotes the period of the *application*. The deadline D of the *application* is equal to its period. We adopt the same assumption as [7], [8], [11], [16] and assume that the worst case execution time (WCET) of each task T_i with a specific L2 cache size can be determined. We use w_{ij} to denote the WCET of task $T_i \in V$ with j ways L2 cache allocated and $W_i = \{w_{i1}, w_{i2}, ..., w_{is}\}$ to denote the WCET profile of task T_i, where s is the total number of ways in the L2 cache (cache capacity). Timing predictability is highly desirable for safety-related applications. In this paper, we consider a periodic time-triggered non-preemptive scheduling policy. We use R to denote the set of the profiles for all tasks in *applications set* A. A task profile $r_i \in R$ is defined as a tuple $r_i =< W_i,\, s_i,\, h_i,\, d_i >$, where s_i, h_i, d_i are respectively the start time, period, and deadline of T_i. The start time s_i is an unknown variable, which is determined by *scheduling* S. The tasks belonging to the same *application* share the same period and deadline.

C. Energy Model

The energy dissipation of cache subsystem comprises of dynamic energy E_{dyn} and static energy E_{sta} [20]: $E_{cache} = E_{dyn} + E_{sta}$. The dynamic energy dissipation E_{dyn} originates from cache accesses and cache misses:

$$E_{dyn} = N_{access} \cdot E_{hit} + N_{miss} \cdot E_{miss} \tag{1}$$

where N_{access} and N_{miss} are the number of cache accesses and misses, respectively. The cache access energy E_{hit} is constant according to cache specification. E_{miss} represents the energy dissipation of a cache miss and is calculated as:

$$E_{miss} = E_{memaccess} + E_{\mu P\_stall} + E_{block\_fill} \tag{2}$$

where $E_{memaccess}$ is the energy dissipation for accessing the off-chip memory, $E_{\mu P\_stall}$ is the energy dissipation when the core is waiting for data from the off-chip memory, and E_{block_fill} is the energy for filling the fetched data to the cache block. The static energy E_{sta} can be computed as $E_{sta} = P_{sta} \cdot t$, where P_{sta} is the static power consumption of cache and t is the total execution time. The data for E_{hit}, E_{block_fill}, and P_{sta} with a given cache specification can be obtained using simulation tools like CACTI [5]. The access and miss numbers N_{access} and N_{miss} are obtained using SimpleScalar [13]. The value for $E_{memaccess}$ and $E_{\mu P\_stall}$ can be obtained from memory and processor specification [20].

D. Problem Statement

Given an *applications set* A with task profile R, a multicore architecture P with s-way associative shared L2 cache, and task mapping $\Pi\{T \rightarrow P\}$, our goal is to find an optimal task-level cache partition CP and feasible scheduling S so that the energy of the cache system E_{cache} is minimized while guaranteeing the timing constraints of all applications as well as the cache capacity constraints.

IV. Motivation

Consider two applications $J_1 = \{T_1, T_2\}$ and $J_2 = \{T_3, T_4, T_5\}$ with dependencies $\{T_1 \rightarrow T_2\}$ and $\{T_3 \rightarrow T_5, T_4 \rightarrow T_5\}$, respectively. For simplicity, we assume J_1 and J_2 have the same period of $H = 8ms$. The MPSoC has a dual-core architecture $P = \{p_1, p_2\}$. A 4-ways cache is shared by both cores. The task profiles under different cache configurations are listed in Tab. I. We compare three different cache partition schemes, namely the a even partitioning scheme, the static core-based partition in [16], and a task-based partitioning scheme. The schedule and the corresponding energy consumption of the three approaches are shown in Fig. 2.

From Fig. 2, we can see that even partitioning has the highest energy consumption. The reason is that, although tasks

TABLE I
Task profiles for different cache setting.

		T_1	T_2	T_3	T_4	T_5
Mapping		p_1	p_2	p_2	p_1	p_2
WCET	way1	5	3	5	3	2.9
	way2	4	2	4.5	1.5	2.6
(ms)	way3	2	1.5	2	1	2
	way4	2	1	2	1	2
Energy	way1	9	12	13	14	9
	way2	6	6	11	14	8
(uJ)	way3	4	5	8	7	6
	way4	3	4	3	6	7

T_1 and T_3 could potentially consume much less energy if more ways of the cache are granted, the pre-allocation of cache excludes this possibility. The static core-based scheme, shown in Fig. 2(b), consumes less energy, because T_1 can run in a more energy-efficient three-way cache setting. Nevertheless, it still some limitations. As shown in the figure, by assigning three ways to core p_1, there is only one way left for core p_2. Task T_3 must unfortunately run in the inefficient one-way configuration. In contrast, the task-based scheme can overcome this drawback. As shown in Fig. 2(c), from time 0 to 3, no task is scheduled for core p_2. Therefore, all four ways are assigned to tasks T_4 and T_1. After T_1 finishes, all these four ways are reassigned to task T_3 on core p_2, since no task is running on core p_1 between time 3 to 5. After T_3 finishes, the cache is shared by both cores again. In this case, the task-based scheme achieves the most energy efficient execution of the tasks.

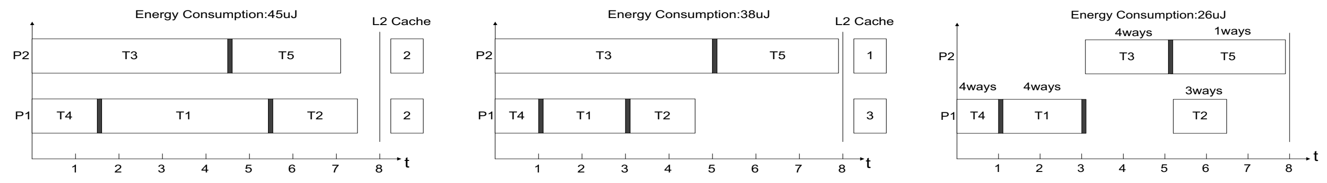
The example indicates that the cache reconfiguration is closely correlated to the schedule of the tasks. Although assigning a larger portion of cache to a task can reduce its execution time, we can not do it greedily due to other constraints. For example, as shown in Fig. 2(c), task T_2 can only be executed after $T3$ since there is no available cache. T_2 and $T5$ are scheduled to execute concurrently due to their deadline constraint, which restricts T_2 from using all 4 ways to further reduce the energy consumption. To tackle this correlation, more sophisticated method is needed to systematically derive the cache reconfigurations as well as the schedule. The next section presents our solution.

V. Proposed Approach

This section presents our mixed integer linear programming (MILP) approach for cache partitioning and task scheduling. We start with an MILP formulation that focuses only on the scheduling problem. Then, the constraints of cache capacity is integrated. Based on the observation that the MILP formulation may suffer from the state explosion, we develop a refinement, the so-called unified resource demand function (URDF) that captures the cache demand of every task and effectively models the interference between tasks, to reduce the exploration space of the formulation.

A. Time-Triggered Task Scheduling

In this paper, we consider time-triggered non-preemptive schedule. For each task T_i with the profile $< W_i,\, s_i,\, h_i,\, d_i >$,



(a) Even partition (b) Core-based partition (c) Task-based partition

Fig. 2. Schedules and energy consumption.

the k-th instance of task T_i starts at $s_i + k \cdot h_i$. W_i contains the WCETs of the task with different cache configurations. We use a set of binary variables c_{ij} to describe the amount of cache allocated to the task T_i: $c_{ij} = 1$ if exactly j cache ways are allocated to T_i and $c_{ij} = 0$ otherwise. In this case, the actual WCET of T_i can be obtained as $\sum_{j=1}^{s} c_{ij} w_{ij}$, where s is the total number of ways of the shared cache. To formulate the scheduling problem by means of MILP, we have to cope with the task dependency, deadlines, and non-preemption. We present our formulation as follows.

Let ξ denote the overheads for cache reconfiguration and task switch. The data dependency $T_j \to T_i$ requires the start time of T_i to be no earlier than the finish time of T_j plus the switching overhead:

$$s_j + \sum_{k=1}^{s} c_{jk} w_{jk} + \xi \le s_i \tag{3}$$

For deadline constraint, task T_i has to finish no later than its deadline:

$$s_i + \sum_{k=1}^{s} c_{ik} w_{ik} \le d_i \tag{4}$$

The non-preemptive constraint requires that any two tasks mapped to the same core must not overlap in time. Let binary variable denote the execution order of task T_i and T_j: $z^{ij}_{p\widetilde{p}} = 1$ if the i-th instance of task T_p finishes before the start of j-th instance of $T_{\widetilde{p}}$, and 0 otherwise. H_r and $H_{p\widetilde{p}}$ denote the hyper-period of all tasks and the hyper-period of only task T_p and $T_{\widetilde{p}}$ (i.e., LCM of periods of T_p and $T_{\widetilde{p}}$), respectively. $TS(T_p)$ denotes the set of tasks that are mapped to the same core as T_p does. The non-preemption constraint can thereby be expressed as follows.

$\forall T_p, T_{\widetilde{p}} \in TS(T_p), i = 0, ..., (\frac{H_{p\widetilde{p}}}{h_p} - 1), j = 0, ..., (\frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}} - 1)$:

$$i \cdot h_p + s_p + \sum_{k=1}^{s} c_{pk} w_{pk} - (1 - z^{ij}_{p\widetilde{p}}) H_r + \xi \le j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} \tag{5}$$

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} - z^{ij}_{p\widetilde{p}} H_r + \xi \le i \cdot h_p + s_p \tag{6}$$

The constraints (5) and (6) ensure that either the instance of T_p runs strictly before the instance of $T_{\widetilde{p}}$, or vice verse.

B. Cache Partitioning Constraints

The constraints described above guarantee a valid time-triggered schedule. The next step is to add the cache partitioning constraints. The goal here is to guarantee the feasibility of cache partitioning, i.e., at any point in time, the sum of cache ways allocated to the tasks currently being executed does not exceeds the cache capacity. In other words, we must avoid *cache overflow*. Before presenting the formulation, we state following lemma.

Lem. 1: If the cache does not overflow at start instant of any task within one hyper-period, the cache never overflows.

Proof: Note that the amount of cache allocated to a task is constant during its execution interval. It acquires the resources at the start instant and releases the resources at the finish instant. Hence, cache overflow will not occur if the available resources fulfill the requirement of tasks at its beginning. ■

From Lem. 1, we know only a finite number of time instants, i.e., at the start of any task, need to be checked for cache overflow. For a specific task T_p, we have to gather all tasks that overlap with it and inspect the total cache demands. Let $T_{\widetilde{p}} \notin TS(T_p)$ be a task running on a different core as T_p. The timing relationship between T_p and $T_{\widetilde{p}}$ can have three possibilities: 1) T_p starts during the execution of $T_{\widetilde{p}}$, i.e., the two tasks overlap in time (or in other words $T_{\widetilde{p}}$ *interferes* T_p), as shown in Fig. 3(a); 2) Task $T_{\widetilde{p}}$ starts after task T_p starts (Fig. 3(b)). In this case, the interference occurs if the start time of $T_{\widetilde{p}}$ is earlier than the finish time of T_p. 3) Task $T_{\widetilde{p}}$ ends before task T_p starts (Fig. 3(c)), i.e., no overlap in time.

Two binary variables are used to describe the three scenarios above. The variable $x^{ij}_{p\widetilde{p}}$ is 1 if the start time of i-th instance of T_p is later than the start time of the j-th instance of $T_{\widetilde{p}}$, and 0 otherwise. The variable $y^{ij}_{p\widetilde{p}} = 1$ if the start time of i-th instance of T_p is earlier than the finish time of j-th instance of $T_{\widetilde{p}}$ ends. Based the above definitions, we define the following constraints.

$\forall T_p, T_{\widetilde{p}} \notin TS(T_p), i = 0, ..., (\frac{H_r}{h_p} - 1), j = 0, ..., (\frac{H_r}{h_{\widetilde{p}}} - 1)$:

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} - (1 - x^{ij}_{p\widetilde{p}}) H_r \le i \cdot h_p + s_p \tag{7}$$

$$i \cdot h_p + s_p - x^{ij}_{p\widetilde{p}} H_r < j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} \tag{8}$$

$$i \cdot h_p + s_p - (1 - y^{ij}_{p\widetilde{p}}) H_r \le j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} \tag{9}$$

$$j \cdot h_{\widetilde{p}} + s_{\widetilde{p}} + \sum_{k=1}^{s} c_{\widetilde{p}k} w_{\widetilde{p}k} - y^{ij}_{p\widetilde{p}} H_r < i \cdot h_p + s_p \tag{10}$$

The constraints (7-10), cover the interference scenarios in Fig. 3, depending on the combination of $x^{ij}_{p\widetilde{p}}$ and $y^{ij}_{p\widetilde{p}}$.

- $x^{ij}_{p\widetilde{p}} = 1$ and $y^{ij}_{p\widetilde{p}} = 1$: (8) and (10) are trivially satisfied while (7) and (9) restrict T_p and $T_{\widetilde{p}}$ to be overlapped. This corresponds to the scenario in Fig. 3(a).
- $x^{ij}_{p\widetilde{p}} = 0$ and $y^{ij}_{p\widetilde{p}} = 1$: (7) and (10) are trivially satisfied while (8) and (9) constraint the execution order of the two tasks to be the scenario in Fig. 3(b). Still, as mentioned before, two possibilities could occur, i.e., T_p interferes

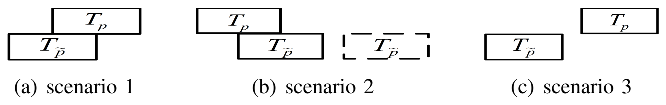


Fig. 3. Timing relationship between two tasks.

$T_{\widetilde{p}}$ [1] and T_p does not interfere $T_{\widetilde{p}}$. The two interference scenarios, that T_p interferes $T_{\widetilde{p}}$ and $T_{\widetilde{p}}$ interferes T_p, are distinguished by the symmetrical binary variable $x^{ji}_{\widetilde{p}p}$ and $y^{ji}_{\widetilde{p}p}$.

- $x^{ij}_{p\widetilde{p}} = 1$ and $y^{ij}_{p\widetilde{p}} = 0$: (8) and (9) are trivially satisfied while (7) and (10) restrict the execution order to be the scenario shown in Fig. 3(c).

Note that $x^{ij}_{p\widetilde{p}}$ and $y^{ij}_{p\widetilde{p}}$ can not be 0 at the same time, due to contradiction between (8) and (10). Based on above analysis, we know $x^{ij}_{p\widetilde{p}}+y^{ij}_{p\widetilde{p}}-1 \in \{0,1\}$ and use the term $x^{ij}_{p\widetilde{p}}+y^{ij}_{p\widetilde{p}}-1$ in the MILP formulation to indicate whether $T_{\widetilde{p}}$ interfere T_p, i.e, whether the scenario in Fig. 3(a) occurs.

Another related constraint is that each task must have exactly one cache configuration.

$$\sum_{k=1}^{s} c_{ik} = 1 \tag{11}$$

Based on Lem. 1, we can now formulate the constraint to guarantee feasibility of cache partitioning. At the start time of a task T_p, the total cache demand consists of the parts from T_p itself and the tasks that interfere with T_p, computed by $\sum_{k=1}^{s} c_{pk} \cdot k$ (for T_p) and $(x^{ij}_{p\widetilde{p}} + y^{ij}_{p\widetilde{p}} - 1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$(for interference task), respectively. The term $(x^{ij}_{p\widetilde{p}} + y^{ij}_{p\widetilde{p}} - 1)$ indicates whether $T_{\widetilde{p}}$ interferes with T_p. Thus, we have the following constraint.
$\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

$$\sum_{k=1}^{s} c_{pk} \cdot k + \sum_{T_{\widetilde{p}} \notin TS(T_p), j=0}^{j=\frac{H_r}{h_{\widetilde{p}}}-1} (x^{ij}_{p\widetilde{p}} + y^{ij}_{p\widetilde{p}} - 1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k \le s \tag{12}$$

One may notice that there are quadratic items in (12), i.e., $(x^{ij}_{p\widetilde{p}}+y^{ij}_{p\widetilde{p}}-1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. However, we can transform this quadratic term into a set of linear constraints using Lem. 2. Here, we define an intermediate variable $t^{ij}_{p\widetilde{p}} = (x^{ij}_{p\widetilde{p}} + y^{ij}_{p\widetilde{p}} - 1)\sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. and can obtain two linear conditions $0 \le \sum_{k=1}^{s} c_{\widetilde{p}k} k \le s$ from (11) and $(x^{ij}_{p\widetilde{p}} + y^{ij}_{p\widetilde{p}} - 1) \in \{0,1\}$.

Lem. 2: Given a constant $s > 0$ and two constraint spaces $P_1 = \{[t,b,x] | t = b \cdot x,\ 0 \le x \le s,\ b \in \{0,1\}\}$ and $P_2 = \{[t,b,x] | 0 \le t \le b \cdot s,\ t \le x,\ t - b \cdot s - x + s \ge 0,\ b \in \{0,1\}\}$, then $P_1 \rightleftharpoons P_2$

Proof: $\mathbf{P_1 \Rightarrow P_2}$: We obtain $0 \le t \le b \cdot s$ according to $t = bx$ and $0 \le x \le s$. From $b \in \{0,1\}$ and $t = bx$, we have $t \le x$. Based on $0 \le x \le s$ and $b \in \{0,1\}$, we can obtain $(b-1)(x-s) \ge 0$. Hence, $t - b \cdot s - x + s \ge 0$ holds. $\mathbf{P_2 \Rightarrow P_1}$: If $b = 0$ holds, we can prove that $t = 0$ and $0 \le x \le s$ according to the definition of P_2. If $b = 1$ holds, we can obtain $0 \le t = x \le s$ from P_2. Thus, $P_2 \rightleftharpoons P_1$. ■

[1]Note that T_p interferes $T_{\widetilde{p}}$ and $T_{\widetilde{p}}$ interferes T_p are two different scenarios.

Up to now, we have presented the formulation for the task scheduling and cache partitioning. To minimize the energy consumption of the cache subsystem in one hyper-period, the following objective is used:

$$E = \sum_{\forall T_i} \frac{H_r}{h_i} \sum_{j=1}^{s} c_{ij} E^{ij}_{cache} \tag{13}$$

where s and E^{ij}_{cache} represent the cache capacity (in the number of ways) and the energy consumption of cache subsystem of task T_i under j-way cache configuration, respectively. The energy profile of each task T_i under different cache configuration can be obtained using the cache energy model depicted in Section III-C.

C. MILP Formulation Refinement

The formulation described in Section V-B utilizes $3Q_0$ variables to model the interference between tasks at each checking instant (i.e, variable $x^{ij}_{p\widetilde{p}}$, $y^{ij}_{p\widetilde{p}}$, and $t^{ij}_{p\widetilde{p}}$). Here, Q_0 is the number of task instances that may interfere with the task under consideration, i.e., $Q_0 = \sum_{T_{\widetilde{p}} \notin TS(T_p)} (\frac{H_r}{h_{\widetilde{p}}})$. Since the task T_p has $\frac{H_r}{h_p}$ instances, the total amount of variables can be calculated as:

$$V_0 = 3 \sum_{\forall T_p} \sum_{T_{\widetilde{p}} \notin TS(T_p)} \frac{H_r}{h_p} \frac{H_r}{h_{\widetilde{p}}} \tag{14}$$

As it can be seen, the total number of variables used increases quadratically with the number of tasks in the application, resulting in dramatically increased exploration space for the MILP. To maintain the scalability of the approach, it is important to develop techniques that can reduce the exploration space. Here, we propose a novel approach based on the concept of Unified Resource Demand Function (URDF). URDF models the resource demand of task in the time domain. For task T_p with start time s_p and execution time e_p, the cache demand at instant t can be defined as:

$$URDF(t, T_p) = \left\lfloor \frac{t - s_p}{h_p} \right\rfloor + 1 - \left\lceil \frac{t - s_p - e_p}{h_p} \right\rceil \tag{15}$$

The URDF above indicates that T_p requires the cache only in interval $[s_p + i \cdot h_p,\ s_p + e_p + i \cdot h_p]$. The URDF has several mathematic properties that are beneficial to model the interference between tasks.

Prop. 1: $URDF(t, T_p) \in \{0, 1\}$. The URDF is 1 is the task T_p requires the cache at time instant t and 0 otherwise.

Prop. 2: $URDF(t, T_p) = URDF(mod(t, h_p), T_p)$.

Prop. 3: Define intermediate variables $X_{t,T_p} = \left\lfloor \frac{t-s_p}{h_p} \right\rfloor$ and $Y_{t,T_p} = \left\lceil \frac{t-s_p-e_p}{h_p} \right\rceil$, then UDRF could be linearized as $UDRF(t, T_p) = X_{t,T_p} + 1 - Y_{t,T_p}$ with two extra constraints $\frac{t-s_p}{h_p} - 1 < X_{t,T_p} \le \frac{t-s_p}{h_p}$ and $\frac{t-s_p-e_p}{h_p} \le Y_{t,T_p} < \frac{t-s_p-e_p}{h_p} + 1$.

Using URDF to model the cache demand, we can reformulate constraint (12) as following.
$\forall T_p, i = 0, ..., (\frac{H_r}{h_p} - 1)$:

$$\sum_{k=1}^{s} c_{pk} \cdot k + \sum_{T_{\widetilde{p}} \notin TS(T_p)} UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k \le s \tag{16}$$

Similar to the linear procedure of (12), we define intermediate variable $a^i_{p\widetilde{p}} = UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$. According to Prop. 1 and Prop. 3, $UDRF(s_p + i \cdot h_p, T_{\widetilde{p}})$ could be linearized as $X_{s_p+i \cdot h_p, T_{\widetilde{p}}} + 1 - Y_{s_p+i \cdot h_p, T_{\widetilde{p}}}$ with $(X_{s_p+i \cdot h_p, T_{\widetilde{p}}} + 1 - Y_{s_p+i \cdot h_p, T_{\widetilde{p}}}) \in \{0, 1\}$. Based on Lem. 2, the non-linear item $UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) \sum_{k=1}^{s} c_{\widetilde{p}k} \cdot k$ in (16) can also be linearized.

The advantage of this reformulation is that we do not need to have separate variables and constraints for every instance of task $T_{\widetilde{p}} \notin TS(T_p)$, resulting in significant reduction of the number of variables and the exploration space. After the refinement, we just need $3Q_1$ variables to model the inter-core interference at a specific checking instant, where Q_1 is the cardinality of the set $T_{\widetilde{p}} \notin TS(T_p)$, i.e. $Q_1 = |\{T_{\widetilde{p}} | T_{\widetilde{p}} \notin TS(T_p)\}|$. Moreover, by applying Prop. 2, we can further reduce the exploration space. We calculate $i_{p\widetilde{p}} = \frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}}$ for each task $T_{\widetilde{p}} \notin TS(T_p)$. Based on Prop. 2, we have $UDRF(s_p + i \cdot h_p, T_{\widetilde{p}}) = UDRF(s_p + mod(i, i_{p\widetilde{p}}) \cdot h_p, T_{\widetilde{p}})$. It indicates that $UDRF$ at checking instant $s_p + mod(i, i_{p\widetilde{p}}) \cdot h_p$ can be reused at checking instant $s_p + i \cdot h_p$. In this case, we only need $i_{p\widetilde{p}}$ URDFs to model the interference between T_p and $T_{\widetilde{p}}$. Since each URDF needs 3 variables in the MILP formulation, the total amount of variable is computed as:

$$V_1 = 3 \sum_{\forall T_p} \sum_{T_{\widetilde{p}} \notin TS(T_p)} \frac{H_{p\widetilde{p}}}{h_{\widetilde{p}}} \tag{17}$$

Compared to (14), we can see the number of variables is significantly reduced after the refinement.

VI. Performance Evaluations

In this section, we demonstrate the effectiveness of our approach. We use SimpleScalar [13] for the simulation and CACTI [5] for collecting the energy parameters of the L2 cache. The CPLEX [6] solver is used to solve the MILP problems. All experiments are conducted on a computer with 2.3GHz Intel CPU and 4GB memory.

A. Experiment Setup

To evaluate the effectiveness of our approach, we consider five task graphs in our experiment: three FFT benchmarks, an Adaptive Cruise Control (ACC) application, and an AES encryption application. The ACC application from Autofocus [1] is an optional cruise control system for road vehicles. It consists of 8 tasks. The AES from CHStone [19] is an application with 6 tasks for the encryption of electronic data. The task graphs for FFT is obtained based on the implementation from MiBench [9]. By employing Cooley-Tukey algorithm [10], N-point FFT can be split into two parallel $\frac{N}{2}$-point FFT. We modified the FFT benchmark in MiBench into three different task graphs with 4 tasks, 7 tasks, and 10 tasks, respectively. The task graph with 4 tasks is obtained by splitting N-point FFT into two $\frac{N}{2}$ FFT tasks. The graph with 7 tasks is obtained by splitting N-point FFT into one $\frac{N}{2}$ FFT tasks and two $\frac{N}{4}$ FFT tasks. Similarly, we construct the graph with 10 tasks by splitting N-point FFT into four $\frac{N}{4}$ FFT tasks. For simplicity, FFT-N-M denotes task graph with M tasks for N-point FFT.

TABLE II
Task graph sets.

	Task graph	Period[ms]	Task#
Set1	FFT-2048-4,FFT-4096-7 FFT-8192-10	50,100,200	21
Set2	FFT-256-4,ACC,AES	12,4,2	16
Set3	FFT-2048-7 FFT-1024-4,ACC	40,20,5	19
Set4	FFT-512-4,AES FFT-1024-4	18,3,36	15
Set5	FFT-1024-10,AES	16,2	14
Set6	FFT-2048-10 AES,ACC,FFT-512-7	40,5,10,20	29

We consider six combinations of these applications. Details of the combinations are shown in Tab. II.

The SimpleScalar cycle-accurate architecture simulation platform [13] is used for our experiment. We modified the cache model in SimpleScalar to support way-based partitioning. The number of cache accesses and misses, as well as the execution time (in cycles), under different cache configurations are obtained using SimpleScalar. The energy parameters of the L2 cache are collected from CACTI [5] with 60 nm technology and [20]. In this paper, we use a 4-core architecture where each core runs at 500 MHz. Private L1 data and instruction caches are set to 4KB and 2KB, respectively. The shared L2 cache is configured to 32KB with 32-byte line size and 8-way associativity. The latencies of L1 cache, L2 cache, and the main memory are set to 1, 10, and 100 cycles, respectively.

B. Results

In this paper, we compare three approaches , i.e., equal partitioning (`EQUAL`), the core-based approach (`CORE-OPT`) from [16], and our approach (`TASK-OPT`). Fig. 4 shows the energy consumption of the approaches normalized w.r.t `EQUAL`. As it can be seen, our approach can on average achieve 40% and 33% energy savings compared to `EQUAL` and `CORE-OPT`, respectively.

Another interesting issue is the impact of the cache size to the effectiveness of our approach. We vary the cache size from 32K to 512K with fixed cache sets and line configuration, i.e., the number of cache ways varies from 8 to 128. Fig. 5 illustrates the results where all values are normalized to `CORE-OPT`. From the figure, we can make the following observations: (1) Our approach can significantly reduce the energy consumption (up to 75%) when the cache size is small. This indicates that our approach is especially useful for architectures with limited resources. (2) As cache capacity increases, the achievable energy saving decreases. This is caused by the fact that the tasks may saturate with the amount of cache allocated, i.e. when the cache allocated to a task exceeds a threshold, the benefit of assigning even more cache is very limited, since the cache behavior, including the number of cache accesses and misses, will be almost identical. In other words, with larger cache capacity, the scheduling problem becomes much easier and the performance gap between `CORE-OPT` and `TASK-OPT` becomes smaller.

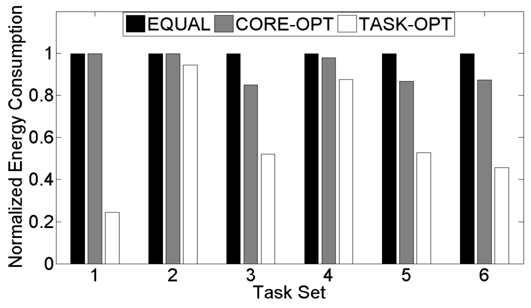


Fig. 4. Cache hierarchy energy reduction.

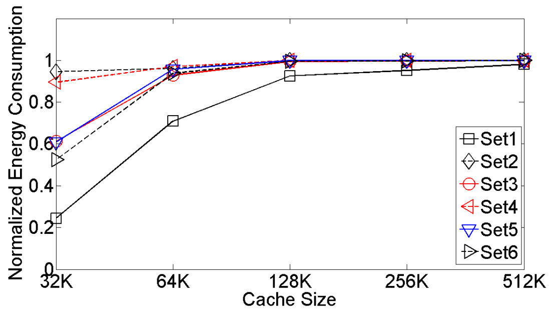


Fig. 5. Normalized energy consumption of the six task sets with different cache sizes.

Nevertheless, our approach outperforms `CORE-OPT` in all cases. (3) Although the performance of `CORE-OPT` and `TASK-OPT` eventually converges with increasing cache size, the speed of convergence depends highly on the complexity of the application. This is because larger tasks need also more cache to enter the saturation state. For example, as shown in the figure, the performance for task set 1 converges much slower due to the high complexity of the FFT tasks. This indicates that our approach is more beneficial to handle future applications with increasing complexity.

In the end, we conduct experiments to show the efficiency of our refinement technique. We adopt the same examples as above using 32KB cache configuration. We compare the number of variables in the MILP and the solving time. Fig. 6 shows the results normalized to the baseline approach without refinement. Note that, for the task set 6, the baseline approach fails to finish within the time budget of 8 hours whereas the refined approach finishes in about 15 minutes. Moreover, the baseline approach needs 14270 variables in the formulation while the refined approach only needs 4139, which results in 71% reduction of the number of variables for task set 6. In Fig. 6, we can observe that the two curves follow the same trend. On average, the refinement achieves 45% reduction on the computation time and 67% on the number of variables.

VII. Conclusion

This paper presents an approach to minimize energy consumption of the cache subsystem by partitioning the cache on task-level. We propose an integrated solution for joint task scheduling and cache partitioning based on mixed integer linear programming. Our approach generates task-level

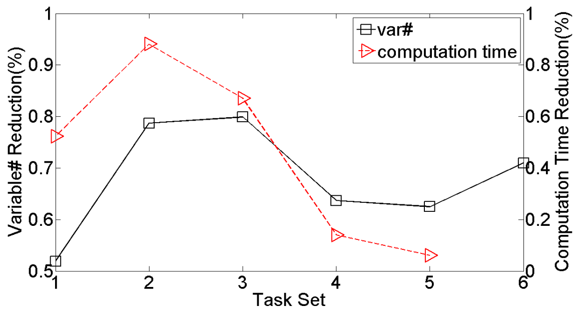


Fig. 6. Numbers of variables and computing time for six different task sets.

cache configurations as well as time-triggered non-preemptive schedule at compilation time, which minimizes the energy consumption of cache subsystem while guaranteeing the timing constraints. Besides, we develop a novel technique that can significantly reduce the exploration space of MILP. Experiment results show that our approach can achieve 33% energy savings compared to existing cache partitioning approaches.

References

[1] AutoFOCUS3. `http://af3.fortiss.org/`.
[2] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. MICRO, 1999.
[3] ARM Cortex-A15 serious. `http://www.arm.com/products`.
[4] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. RTCSA, 2008.
[5] CACTI. `http://www.hpl.hp.com/research/cacti`.
[6] IBM ILOG CPLEX. `http://www.ibm.com/software/`.
[7] X. Fu, K. Kabir, and Xiaorui. Cache-aware utilization control for energy efficiency in multi-core real-time systems. ECRTS, 2011.
[8] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. EMSOFT, 2009.
[9] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. WWC, 2001.
[10] M. B. Henry and L. Nazhandali. Hybrid super/subthreshold design of a low power scalable-throughput fft architecture. HiPEAC, 2009.
[11] T. Liu, Y. Zhao, M. Li, and C. J. Xue. Joint task assignment and cache partitioning with cache locking for wcet minimization on mpsoc. *Journal of Parallel and Distributed Computing*, 2011.
[12] R. Reddy and P. Petrov. Cache partitioning for energy-efficient and interference-free embedded multitasking. *ACM Transactions on Embedded Computing Systems*, Mar. 2010.
[13] SimpleScalar LLC. `http://www.simplescalar.com`.
[14] OpenSPARC. `http://www.opensparc.net/`.
[15] W. Wang, P. Mishra, and A. Gordon-Ross. Dynamic cache reconfiguration for soft real-time systems. *ACM Transactions on Embedded Computing Systems*, 2012.
[16] W. Wang, P. Mishra, and S. Ranka. Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems. DAC, june 2011.
[17] S.-H. Yang, B. Falsafi, M. D. Powell, K. Roy, and T. N. Vijaykumar. An integrated circuit/architecture approach to reducing leakage in deep-submicron high-performance i-caches. HPCA, 2001.
[18] S.-H. Yang, B. Falsafi, M. D. Powell, and T. N. Vijaykumar. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. HPCA, 2002.
[19] S. H. Yuko Hara, Hiroyuki Tomiyama and H. Takada. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 2009.
[20] C. Zhang, F. Vahid, and W. Najjar. A highly configurable cache for low energy embedded systems. *ACM Transactions on Embedded Computing Systems*, 2005.