Cache Allocation for Fixed-Priority Real-Time Scheduling on Multi-Core Platforms

Gustavo A. Chaparro-Baquero*, Soamar Homsi*, Omara Vichot*, Shaolei Ren*, Gang Quan* and Shangping Ren†
*Electrical and Computer Engineering Department. Florida International University (FIU). Miami, FL, 33174. U.S.A.
e-mail: {gchap002,shoms001,ovich001,sren,gaquan}@fiu.edu
†Department of Computer Science. Illinois Institute of Technology (IIT). Chicago, IL, 60616. U.S.A
e-mail: ren@iit.edu

***Abstract*—The increased resource sharing on multi-core platforms has posed significant challenges on the predictability of real-time systems. Cache memory partitioning has proven to be one of the most effective methods to improve the predictability and also the schedulability of real-time systems. In this paper, we study how to allocate cache memory of a multi-core platform when scheduling fixed-priority hard real-time tasks. As the bounded worst-case execution time (WCET) of a real-time task varies with its cache allocation, the challenges of this problem are twofold: how to judiciously allocate the cache memory among all real-time tasks and how to map real-time tasks to each core to improve the schedulability. To address these challenges, we develop an approach that takes into consideration not only the WCET variations with cache allocations but also the task period relationship and thus can significantly improve the schedulability of real-time tasks. Our simulation results, based on the SPEC CPU2000 benchmarks suite, show that our approach can increase the schedulability of real-time tasks up to four times when compared to other similar scheduling mechanisms.**

I. Introduction

Real-time systems used to be developed on single-core platforms, but a high increase in processing demands has led the industry to develop them on multi-core platforms in the past few years. The increase in processing demands comes from the fact that today's real-time applications, like intelligent cruise control and unmanned vehicles control, process significant amounts of sensor data with real-time constraints. These applications not only generate large amounts of I/O workloads, but also become more and more memory intensive. Thus, multi-core architectures have become an attractive option for developing real-time systems due to their potential to handle the newly increased workloads.

In the meantime, however, the large inter-task interferences due to increased resource sharing (such as shared buses and memory) on multi-core platforms have severely undermined the predictability of real-time systems [1], [2]. For the sake of scalability, flexibility, and to deal with power limitation in the era of "dark silicon," it has become mainstream to group multiple cores sharing a local cache memory [3].

The sharing of local cache memory helps to improve the average case execution time of each task, but can be hazardous to the estimation of the worst-case execution time (WCET). One major problem in estimating the WCET bounds on multi-core systems is the unpredictability of the workload on other cores. Therefore, the number of memory accesses, locations in time, and bus loads originated from other concurrent tasks are difficult to determine precisely [2]. To assume the worst case scenario for each factor can be extremely pessimistic and nullifies the extra computational capacity of the multi-core platforms in the design of real-time systems.

Since a major source of pessimism in WCET estimation comes from shared cache memories, cache memory partitioning has proven to be one of the most effective methods to improve the predictability and schedulability of real-time systems [4]–[9]. This method partitions cache memory among programs and cores to reduce cache contention. By isolating real-time task memory accesses, cache memory partitioning can avoid or considerably reduce the inter-task interferences, and therefore reduce the uncertainty when bounding the WCET and improve the core utilization.

In this paper, we are interested in studying the problem of how to allocate the cache memory that is accessible by multiple processing cores when scheduling fixed-priority real-time tasks based on the rate monotonic scheduling (RMS) policy. The fixed-priority multi-core partitioned scheduling scheme is one of the most commonly used scheduling mechanisms for real-time system design [10], due to its advantage of better predictability. Besides, it is supported by almost all real-time operating systems available on the market due to its low overhead and simplicity in implementation, and it is still the method of choice in industry. We assume that each real-time task will be executed on a dedicated processing core, and its WCET, for a specified cache size, can be estimated beforehand using strategies such as those presented in [11].

Since the WCET of a real-time task varies with its cache allocation, our research problem involves two intertwined problems: *i)* how to allocate the available cache memory partitions among all tasks, and *ii)* how to map each task to a core in the multi-core platform. One simple approach to partition the cache memory is to allocate the cache memory in such a way that it minimizes the *normalized resource usage* [12] —which includes both CPU utilization and memory utilization— for each task. However, the cache allocation

This work is supported in part by the National Science Foundation (NSF) under projects CNS-1423137 and CNS-1018108.

that optimizes the resource usage for a single task does not necessarily optimize that for the entire task set. To map tasks to multiple cores and optimize CPU resource usage is a classical NP-hard problem. While it has been a well-known fact that harmonic tasks can utilize CPU resource more effectively, i.e. with CPU utilization as high as 1 [13], how to take the interplay of cache partitioning, execution time variations and task harmonic relationship into considerations to deal with cache allocation and task mapping in an integrated manner is the challenging problem we want to study in this paper.

We propose two algorithms in this paper. The first algorithm combines two existing works: one based on fast local memory partitioning [12], and the other one, on harmonic-based scheduling [14]. The second algorithm is a more elaborated approach that can judiciously choose the cache size for each task and also exploit task harmonic relationships. Therefore, it can significantly improve the system resource usage and task set schedulability. We use a third party data report of the cache performance for the SPEC CPU2000 benchmarks suite [15] to validate our approaches. The results show that our approach can significantly improve the schedulability of real-time tasks, i.e. up to four times, when compared with other scheduling mechanisms.

II. Preliminary

In this section, we introduce the architecture and the real-time system model used in this paper. We also show an example to motivate our research.

A. Architecture and System Model

The multi-core platform consists of a set of P homogeneous processing cores, denoted as P_k with $k = 1, 2, ..., P$. The cache memory is divided into a finite number of allocation units of the same size called *cache units*. The total number of cache units is denoted as B.

The task set consists of N independent implicit-deadline periodic tasks, denoted as $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$, scheduled according to RMS. Each task τ_i, where $1 \leq i \leq N$, is characterized by its minimum inter-arrival time T_i. A finite number, denoted as m_i, of cache units are assigned privately to a single task τ_i executed by a core of the system, and its WCET varies with m_i, which is denoted as $C_i^{m_i}$. Therefore, the task set Γ may be characterized by a matrix like the one shown Table I. In this table, the rows indicate each task belonging to the task set (four tasks for the example), and each column (except for the last one) indicates the number of assigned cache units m_i to each task $(1 \leq m_i \leq 16)$. The numbers shown in the matrix correspond to each $C_i^{m_i}$ of each task. The last column indicates the period of each task.

Each task $\tau_i \in \Gamma$ is characterized by a CPU-utilization and a memory-utilization. We define $U_i^{m_i}$ as the CPU-utilization of τ_i, where $U_i^{m_i} = C_i^{m_i}/T_i$ and B_i as the Memory-utilization of τ_i where $B_i = m_i/B$. In the same way, we also define the CPU-utilization of a task set Γ as $U(\Gamma) = \sum_{\tau_i \in \Gamma} U_i^{m_i}$, and the total number of cache units used by a task set $q(\Gamma) = \sum_{\tau_i \in \Gamma} m_i$.

Table I: Example of Task Set and the WCET values for different m_i

	m_i																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	T_i
$C_1^{m_1}$	5	5	4	4	3	3	3	3	3	3	3	3	3	3	3	3	10
$C_2^{m_2}$	20	18	10	6	6	6	6	2	1	1	1	1	1	1	1	1	25
$C_3^{m_3}$	10	8	6	6	6	5	5	5	4	4	4	4	4	4	4	4	13
$C_4^{m_4}$	10	9	8	7	6	5	5	5	5	5	5	5	5	5	5	5	25

When task $\tau_j \in \Gamma$ is assigned with a specific value of m_j, we reference its CPU utilization as U_j.

B. Cache Allocation Example

Before we present our algorithms, we first show an example, i.e. Example 1, of a cache allocation problem along with two possible solution methods: the first one, using a previous proposed technique, and the second one, using a simple inspection.

Example 1: Consider a task set consisting of four tasks, as shown in Table I, to be scheduled in a platform consisting of two cores, sharing a cache memory with 16 cache units.

The problem defined in Example 1 has proven to be NP-hard. One solution for this problem, i.e. IBRT-MCI-RMS presented by Chang et al. [16], is to first allocate the cache space that can optimize the resource usage for a single task, and then transform this problem to the traditional bin-packing problem. To this end, they first define a metric, called *normalized resource usage*, to balance CPU and cache resource usage, as shown in the following definition.

Definition 1: The *minimum normalized resource usage* [16] of task $\tau_i \in \Gamma$, denoted as λ_i, is defined as:

$$\lambda_i = \min_{0 \leq m_i \leq B} \left(\frac{U_i^{m_i}}{P} + \frac{m_i}{B} \right) \tag{1}$$

Essentially, the minimum normalized resource usage of a task is the minimum sum of its normalized processor utilization and the normalized cache allocation. With task execution times and cache allocations given, the minimum normalized resource usage of a task can be readily identified. Table II shows the cache allocation results based on this approach. Columns m_i and C_i, and thus U_i are obtained based on Def. 1. Then, IBRT-MCI-RMS sorts tasks in a non-decreasing order with respect to their values of m_i, and packs tasks to cores with utilization bounded by the traditional Liu&Layland upper-bound [17]. In the case of a task set of two tasks, such bound is of 0.83. For this example, as shown in Table II, the total utilization for the subtask set with τ_2 and τ_3 (a value of 0.70) is less than the upper bound. However, the value of total utilization for the subtask set τ_1 and τ_4 (a value of 0.90) is larger than the utilization bound. Therefore, IBRT-MCI-RMS fails to schedule the task set of Example 1.

For the problem defined above, a feasible solution does exist. As shown in Table III, by assigning 3 cache units to τ_1 and 2 cache units to τ_4, τ_1 and τ_4 would decrease their WCETs from 5 to 4 and from 10 to 9, respectively, making the task set comply with the schedulability condition defined by the Liu&Layland upper-bound. Besides, the total number

Table II: Motivation Example Solution Using IBRT-MCI-RMS [16]

τ_i	T_i	C_i	m_i	U_i	Sub Task Set Utilization	Sched. Cond.	Feasibility
2	25	6	4	0.24			
3	13	6	3	0.46	$U(\Gamma_{2,3}) = 0.70$	$U(\Gamma_{2,3}) \overset{?}{\leq} 0.83$	Core 1 - YES
1	10	5	1	0.50			
4	25	10	1	0.40	$U(\Gamma_{1,4}) = 0.90$	$U(\Gamma_{1,4}) \overset{?}{\leq} 0.83$	Core 2 - NO

Table III: Motivation Example Solution by Inspection

τ_i	T_i	C_i	m_i	U_i	Sub Task Set Utilization	Sched. Cond.	Feasibility
2	25	6	4	0.24			
3	13	6	3	0.46	$U(\Gamma_{2,3}) = 0.70$	$U(\Gamma_{2,3}) \overset{?}{\leq} 0.83$	Core 1 - YES
1	10	<u>4</u>	<u>3</u>	<u>0.40</u>			
4	25	<u>9</u>	<u>2</u>	<u>0.36</u>	$U(\Gamma_{1,4}) = $ <u>0.76</u>	$U(\Gamma_{1,4}) \overset{?}{\leq} 0.83$	<u>Core 2 - YES</u>

of cache units used by the task set would be increased from 9 to 12, which is still less than 16. The numbers underlined in Table III represent the changed values from the solution shown in Table II. This shows that, even though IBRT-MCI-RMS allocates cache space to optimize the resource usage (according to λ_i of Def. 1) of a single task, the local optimum solution cannot guarantee that the solution is globally optimal. In addition, it is well-known that period relationship of real-time tasks has a significant impact on their schedulability on a processor [13], [18]. The question is how to take it into consideration in cache allocation and task mapping to improve system resource usage and schedulability of real-time task sets.

III. Simple Harmonic-Based Cache Allocation Approach (HBCA1)

One way to exploit the period relationship among tasks is to simply incorporate the task period into the task mapping phase only. During the cache allocation phase, we can search a local optimal value for the parameters C_i and m_i for each $\tau_i \in \Gamma$ based on the metric λ_i described in Def. 1 [16]. Note that, after λ_i is defined, the WCET for each task is also defined. Then, we can employ the harmonic-based task mapping method (such as the one in [14]) to map tasks to multi-core platforms. We call this approach HBCA1 (Harmonic-Based Cache Allocation 1), which is shown in Alg. 1. In Alg. 1, we assume that all processing cores share the same cache memory. The algorithm can be easily extended to deal with the scenario of when processing cores share multiple cache memories.

While a harmonic task set can be schedulable with total utilization reaching as high as 1, not all tasks are harmonic. Therefore, to better exploit the harmonic relationship among tasks, one critical question is how *harmonic* a task set is. To this end, Fan et al. [14] introduce the concept of *primary sub-harmonic task set* and, based on it, they develop the *harmonic index* to quantify the harmonicity.

Definition 2: [14] Given a task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$ where $\tau_i = (C_i, T_i)$, let $\Gamma' = \{\tau'_1, \tau'_2, ..., \tau'_N\}$ be a harmonic task set with $\tau'_i = (C_i, T'_i)$ and $T'_i \leq T_i$. Then, Γ' is called a *Primary Sub-Harmonic (PSH) task set* of Γ if there exists no harmonic task set $\Gamma'' = \{\tau''_1, \tau''_2, ..., \tau''_N\}$, $\tau''_i = (C_i, T''_i)$ and $T''_i \leq T_i$, such that for $T'_i \leq T''_i$ for all $1 \leq i \leq N$.

Algorithm 1 Simple Harmonic-Based Cache Allocation Approach (HBCA1)

Input: Γ, P, B, WCET Task Matrix
Output: Cache Allocation && Task Partition Results

1: *rem_cacheunits* $= B$ /*Remaining cache units in memory*/
2: *rem_Cores* $= P$ /*Remaining idle cores sharing mem.*/
3: $\Gamma_{TS} = \emptyset$; $\mathcal{P} = \{P_1, P_2, ..., P\}$;
4: **for all** $\tau_i \in \Gamma$ **do**
5: Find m_i such that: $\left(\frac{U_i^{m_i}}{P} + \frac{m_i}{B}\right)$ is minimum; $C_i = U_i^{m_i} \cdot T_i$;
6: **end for**
7: **while** $\Gamma \neq \emptyset$ && $|\mathcal{P}| \neq 0$ **do**
8: Sort τ_i increasing order with respect to T_i;
9: $n = |\Gamma|$; $U_{TS} = -\infty$; $B_{th} = \frac{rem_cacheunits}{rem_Cores}$;
10: **for** $i = 1$ to n **do**
11: Construct Γ' (PSH task set of Γ) using *DCT* [18] with τ_i as base;
12: Sort all $\tau_j \in \Gamma$ in increasing order with respect to $\Delta U_j = U'_j - U_j$;
13: Γ_{k_j} = pick up k_j tasks from Γ such that:
(1) $U(\Gamma'_{k_j}) \leq 1$; (2) $U(\Gamma_{k_j})$ is maximized; (3) $q(\Gamma_{k_j}) \leq B_{th}$;
14: **if** $\{U(\Gamma'_{k_j}) \leq 1\}$ AND $\{U(\Gamma_{k_j}) > U(\Gamma_{TS})\}$ **then** $\Gamma_{TS} = \Gamma_{k_j}$; **end if**
15: **end for**
16: Assign Γ_{TS} *to* $P_k \in \mathcal{P}$; $\mathcal{P} = \mathcal{P} - P_k$; $\Gamma = \Gamma - \Gamma_{TS}$;
17: Recalculate *rem_cacheunits* and *rem_Cores*;
18: **end while**
19: **if** $\Gamma \neq \emptyset$ **then** Return: Γ is not schedulable; **end if**=0

Definition 3: [14] Given a task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$ where $\tau_i = (C_i, T_i)$, let $\mathcal{PSH}(\Gamma)$ denote the set of all PSH task sets for Γ. The *harmonic index*, denoted as $H(\Gamma)$, is defined as:

$$H(\Gamma) = \min_{\Gamma' \in \mathcal{PSH}(\Gamma)} (U(\Gamma') - U(\Gamma)), \tag{2}$$

where $U(\Gamma')$ and $U(\Gamma)$ are the overall system utilizations for Γ' and Γ, respectively.

The lower a task set's harmonic index is, the *closer* it is to one of its primary sub-harmonic task sets and therefore more harmonic. As discussed by Fan et al. [14], one approach to identify sub-harmonic task sets for a given task set is to employ the *DCT* algorithm [18]. In addition, the schedulability of a real-time task set can be predicted based on its sub-harmonic task sets, as stated in the following theorem:

Theorem 1: [18] Let Γ' be a sub-harmonic task set of Γ. Then, Γ is feasible on a single processing unit under RMS, if $U(\Gamma') \leq 1$.

Alg. 1 first determines the local optimal cache allocation based on the metric λ_i described in Def. 1 (lines 4 to 6). Then, it packs tasks that are most harmonic to the reference task (τ_i) in a sub task set Γ_{k_j} and maximizes the task utilization (line 13, condition 2). To prevent "greedy" tasks from hoarding all the available memory cache units, we set a *cache units allocation threshold* (CUAT), i.e. B_{th}, requiring that the total cache units allocated to tasks on the same sub task set Γ_{k_j} should not exceed B_{th} (line 13, condition 3). In our approach, we define B_{th} as the average available cache units for each core. This procedure is repeated by taking each $\tau_i \in \Gamma$ as the reference task (*for loop* line 10). The schedulable task set with the highest utilization, i.e. Γ_{TS}, is allocated to a processing core (line 16). The CUAT is recalculated and the procedure is repeated for the rest of the tasks and cores, until there are no

Table IV: Solution to Example 1 using HBCA1

CHOOSE SUB TASK SET FOR CORE 1												
1	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	ΔT_i	$\Delta \mathbf{U}_i$	$Cum.U_i'$	$Cum.U_i$	
	τ_1	10	5	1	0.50	10	0.50	0	**0**	0.5	0.50	✓
	τ_2	25	6	4	0.24	20	0.30	5	**0.06**	0.8	0.74	✓
	τ_4	25	10	1	0.40	20	0.50	5	**0.10**	1.3	1.14	
	τ_3	13	6	3	0.46	10	0.60	3	**0.14**	1.9	1.60	
2	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	ΔT_i	$\Delta \mathbf{U}_i$	$Cum.U_i'$	$Cum.U_i$	
	τ_3	13	6	3	0.46	13	0.46	0	**0**	0.47	0.46	
	τ_2	25	6	4	0.24	13	0.46	12	**0.22**	0.92	0.70	
	τ_1	10	5	1	0.50	6.5	0.77	3.5	**0.27**	1.69	1.20	
	τ_4	25	10	1	0.40	13	0.77	12	**0.37**	2.46	1.60	
3	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	ΔT_i	$\Delta \mathbf{U}_i$	$Cum.U_i'$	$Cum.U_i$	
	τ_2	25	6	4	0.24	25	0.24	0	**0**	0.24	0.24	
	τ_4	25	10	1	0.40	25	0.40	0	**0**	0.64	0.64	
	τ_3	13	6	3	0.46	12.5	0.48	0.50	**0.02**	1.12	1.10	
	τ_1	10	5	1	0.50	6.26	0.80	3.75	**0.30**	1.92	1.60	
4	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	ΔT_i	$\Delta \mathbf{U}_i$	$Cum.U_i'$	$Cum.U_i$	
	τ_2	25	6	4	0.24	25	0.24	0	**0**	0.24	0.24	
	τ_4	25	10	1	0.40	25	0.40	0	**0**	0.64	0.64	
	τ_3	13	6	3	0.46	12.5	0.48	0.50	**0.02**	1.12	1.10	
	τ_1	10	5	1	0.50	6.25	0.80	3.75	**0.30**	1.92	1.60	
CHOOSE SUB TASK SET FOR CORE 2												
1	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	ΔT_i	$\Delta \mathbf{U}_i$	$Cum.U_i'$	$Cum.U_i$	
	τ_3	13	6	3	0.46	13	0.46	0	**0**	0.46	0.46	
	τ_4	25	10	1	0.40	13	0.77	12	**0.37**	1.23	0.86	
2	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	ΔT_i	$\Delta \mathbf{U}_i$	$Cum.U_i'$	$Cum.U_i$	
	τ_4	25	10	1	0.40	25	0.40	0	**0**	0.40	0.40	✓
	τ_3	13	6	3	0.46	12.5	0.48	0.5	**0.02**	0.88	0.86	✓

more tasks left or no more cores are available in the system (*while loop* line 7).

As an example, Table IV shows the solution to the problem described in Example 1 using HBCA1. The two sections in the table correspond to the procedures to find the sub task sets for Core 1 and Core 2, respectively. Columns labeled as C_i, m_i, T_i corresponds to the WCETs, allocated cache units, and periods of tasks. Columns labeled as T_i', U_i' and U_i show periods and utilizations of tasks in the PSH task sets. Tasks in Table IV are sorted based on ΔU_i. Columns of ΔT_i, ΔU_i are the period and utilization differences between a task with its corresponding task in the PSH task set. Columns of *Cum.* U_i and *Cum.* U_i' are the sums of the values for U_i and U_i' for when each task in the row is added. For example, in the first PSH task set, for τ_1 the *Cum.* $U_i' = 0.5$, for $\tau_1 + \tau_2$ the *Cum.* $U_i' = 0.8$, and for $\tau_1 + \tau_2 + \tau_4$ the *Cum.* $U_i' = 1.3$, which is larger than 1, indicating that only τ_1 and τ_2 can be scheduled together in one core (according to Theorem 1).

At the beginning, there are four tasks in the task set to be scheduled, and therefore the algorithm generates four different PSH task sets, as shown in the four rows of the first section in Table IV. The first one generated is the best candidate to be scheduled in core 1 since the feasible sub-task set (i.e. τ_1, τ_2) has the largest accumulated utilization (i.e. $U(\{\tau_1, \tau_2\}) = 0.74$) among the four. Hence, τ_1 and τ_2 are scheduled to core 1. The algorithm continues allocating the remaining tasks, repeating the process. In this case, the algorithm generates two different PSH task sets. The second row in the second section of Table IV shows that $U(\{\tau_3, \tau_4\}) = 0.86$ and $U'(\{\tau_3, \tau_4\}) = 0.88 \leq 1$. This ensures that τ_3 and τ_4 can be scheduled to core 2.

The complexity of Alg. 1 mainly comes from the loop from lines 10-15 with a complexity of $O(n^2 log\, n)$. Since the loop will be executed for P times, the overall complexity of Alg. 1 is $O(Pn^2 log\, n)$. While Alg. 1 can successfully schedule the task sets in Example 1, one big limitation of this approach is its local optimum cache allocation, i.e. optimum from each task's perspective. In what follows, we develop a more elaborate cache allocation and task scheduling approach that considers the task harmonic relationship.

IV. Enhanced Harmonic-Based Cache Allocation Approach (HBCA2)

In order to increase the schedulability of the system, we propose a second and more elaborate approach. The second approach is called the HBCA2 (Harmonic-Based Cache Allocation 2), and is shown in Alg. 2. It does not allocate cache memory based solely on the relation of WCET and number of cache units for each individual task. Instead, HBCA2 first groups tasks according to their harmonic relationship. Then, it allocates memory cache units to tasks in a way that can decrease the task set CPU utilization the most, when assigned with the same or less number of cache units possible.

The first problem for HBCA2 is to identify the candidate sub-task sets that may be assigned to a single core. Since the harmonic task sets can better utilize CPU resources, one intuitive approach is to employ the harmonic index as defined in Def. 3 and allocate tasks with a high harmonic index to the same core. However, since the cache allocations have not been determined, and thus the WCETs are not available, the harmonic index defined in Def. 3 does not apply. As a result, we use a different harmonic index ($H_t(\tau_i, \tau_j)$) to quantify, for a given task set, how harmonic a task is to a reference task.

Definition 4: Let $\Gamma_j' = \{\tau_1', \tau_2', ..., \tau_N'\}$ be a PSH task set of a task set $\Gamma = \{\tau_1, \tau_2, ..., \tau_N\}$ with $\tau_j' = \tau_j$. The *harmonic index of task* $\tau_i \in \Gamma$ with respect to task $\tau_i' \in \Gamma_j'$, denoted as $H_t(\tau_i, \tau_j)$, is defined as:

$$H_t(\tau_i, \tau_j) = \frac{T_i - T_i'}{T_i}. \tag{3}$$

Note that the harmonic index defined in Def. 4 is independent of its WCET or cache allocation. Therefore, we can construct the PHS task sets and order tasks based on the new harmonic index before the cache allocation is performed. The question becomes how to allocate cache units to the selected tasks with a high degree of harmonic relationship.

We develop an incremental approach for the cache allocation. Specifically, we first set the number of cache units to be 1 (i.e. $m_i = 1$) for each task, i.e. the most unbalanced resource allocation when the CPU utilization is maximized and the memory utilization is minimized for each task (line 4). Tasks with high harmonic index values are grouped into one sub-task set Γ_i', until (*i*) no task can be added to the sub-task set while keeping the task set schedulable; (*ii*) the total cache units are no more than CUAT, i.e. B_{th}, as explained before (line 12).

Since the number of total cache units for the selected tasks is less than B_{th}, an opportunity is presented to allocate more cache units to the selected tasks, i.e. Γ_i'. As these selected

Algorithm 2 Enhanced Harmonic-Based Cache Allocation Approach (HBCA2)

Input: Γ, B, P, WCET Task Matrix
Output: Cache Allocation && Task Partition Results

```
1:  rem_cacheunits = B  /*Remaining cache units in memory*/
2:  rem_Cores = P  /*Remaining idle cores sharing mem.*/
3:  Γ_TS = ∅; 𝒫 = {P_1, P_2, ..., P};
4:  for all τ_i ∈ Γ do m_i = 1; C_i = C_i^1; end for
5:  while Γ ≠ ∅ && |𝒫| ≠ 0 do
6:      Sort τ_i ∈ Γ by the increasing order of T_i;
7:      n = |Γ|; U_TS = −∞; B_th = rem_cacheunits / rem_Cores;
8:      for i = 1 to n do
9:          Construct Γ' (PSH task set of Γ) using DCT [18] with τ_i as base;
10:         Sort τ_j ∈ Γ by the increasing order of H_t(τ_j, τ_i);
11:         step = 1; Γ'_i = ∅;
12:         Γ'_i = pick up the first j tasks listed from Γ such that:
                (1) U(Γ'_i) ≤ 1;    (2) q(Γ'_i) ≤ B_th;
13:         while j ≤ n do
14:             j = j + 1; Γ'_i = Γ'_i + τ_j;
15:             while U(Γ'_i) > 1 && q(Γ'_i) < B_th do
16:                 Find τ_GT ∈ Γ'_i s.t: CRRI(τ_GT, m_GT, step) is max.;
17:                 if τ_GT is unique then
18:                     m_GT = m_GT + step; step = 1; recalculate q(Γ'_i) and U(Γ'_i);
19:                 else
20:                     step = step + 1;
21:                 end if
22:             end while
23:             if U(Γ'_i) > 1 then Γ'_i = Γ'_i − τ_j; end if
24:             if q(Γ'_i) ≥ B_th then break; end if
25:         end while
26:         if U(Γ'_i) > U(Γ_TS) then
27:             if {|Γ_i| > |Γ_TS|} OR {|Γ_i| == |Γ_TS| AND q(Γ'_i) ≤ q(Γ_TS)} then
                Γ_TS = Γ'_i; end if
28:         end if
29:     end for
30:     Assign Γ_TS to P_k ∈ 𝒫; 𝒫 = 𝒫 − P_k; Γ = Γ − Γ_TS;
31:     Recalculate rem_cacheunits and rem_Cores;
32: end while
33: if Γ ≠ ∅ then Return: Γ is not schedulable; end if=0
```

tasks decrease their execution times with more cache units, more tasks can be assigned in the processing core without compromising the schedulability (*while loop* line 13).

To this end, we design a new metric $CRRI(\tau_j)$ (*Combined Resources Ratio Index (CRRI)*) as follows:

Definition 5: Let $C_i^{m_i}$ and $C_i^{m_i+x}$ be the WCETs with respect to the (privately assigned) shared cache size of m_i and m_i+x cache units. The Combined Resources Ratio Index (CRRI) of τ_i, denoted as $CRRI(\tau_i, m_i, x)$, is defined as

$$CRRI(\tau_i, m_i, x) = \frac{\Delta U_i}{\Delta B_i} \quad (4)$$

where $\Delta U_i = (C_i^{m_i} - C_i^{m_i+x})/T_i$ (the decrement in CPU utilization for τ_i) and $\Delta B_i = x/B$ (the increase in memory utilization for τ_i), B is the total number of cache units in a shared cache.

CRRI is essentially a benefit/cost index for cache allocation to a task. A higher CRRI value means that the decrement of WCET of τ_i is larger with a smaller number of extra cache units assigned to it. Thus, the higher the value for *CRRI*, the better the resource usage efficiency. One by one the next tasks in line (according to the harmonic index order) are assigned to Γ'_i (line 14), making the task set unschedulable. Therefore, the number of cache units for the task with the highest *CRRI* value, so called the *Guilty-Task(GT)*, is increased until the task set is schedulable again, i.e. $U(\Gamma'_i) \leq 1$, or the number of total cache units assigned to the task set exceeds B_{th} (*while loop* line 15). This procedure is then repeated until the maximum number of cache units allowed for tasks on each core is reached (line 24). If the next task in line cannot be added to the existing task set, the original cache allocation for the existing task set is recovered (line 23).

The complexity of Alg. 2 mainly comes from the loop from lines 8 to 30. Assuming that in the worst case each core can accommodate n tasks, the complexity of the loop is $O(n^3)$ and the overall complexity of the algorithm is $O(Pn^3)$.

Similar to Alg. 1, Alg. 2 constructs the sub-harmonic task set based on each task using the *DCT* algorithm. As the *DCT* algorithm generates one PSH task set when each τ_i is taken as the reference task, the algorithm comes up with n different sub-task sets to be allocated to a core. These task sets may have different performances in terms of system utilizations, task numbers, and total numbers of cache units, which conflict with each other. To explore all the Pareto optimal solutions may lead to an extremely large search space and is not realistic. In our approach, we adopt a simple metric as follows to choose the best sub-tasks to map to a core: The chosen task set is the one that has the maximum $U(\Gamma'_i)$ value with the highest total number of tasks $|\Gamma'_i|$. If the task numbers are the same, then the one with the smaller total number of used cache units $q(\Gamma'_i)$ wins (lines 26 to 28).

As an example, Table V shows the solution to the problem described in Example 1 using HBCA2. Data is presented in the same way as in Table IV, but tasks in Table V are sorted based on ΔT_i. Unlike HBCA1, algorithm HBCA2 is able to notice that by assigning three extra cache units to τ_4 (values underlined in the table), it is possible to schedule tasks τ_2, τ_3 and τ_4 together on core 1, with a CPU utilization of 0.98 and using 11 memory cache units. Then, τ_1 is scheduled to core 2. Although the algorithm still requires two cores to schedule the task set, it leaves more CPU utilization to be used on core 2 by an additional 5th task. Consequently, we can say that our second approach is able to improve the system resource usage and the schedulability. It is noteworthy to mention that for the first two sub-harmonic task sets generated, the algorithm notices that τ_1 is not schedulable along with τ_3 (using the condition of Alg. 2, line 23). Such unschedulability is shown in the table with the strikethrough text. Then, the algorithm proceeds to try to schedule the next task in the list, i.e. τ_2.

V. Experiments, Analysis and Results

In sections III and IV, two approaches are proposed. It is hard to prove if one dominates the other analytically. Therefore, we use simulation results to study their performance and compare them with related work.

A. SPEC CPU2000 Benchmarks Cache Simulation

In order to test our scheduling approach, we use the data presented in [15], corresponding to the simulation results of the SPEC CPU2000 benchmarks [19] using the Simplescalar

Table V: Solution to Example 1 using HBCA2

CHOOSE SUB TASK SET FOR CORE 1												
1	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	$\Delta\mathbf{T}_i$	ΔU_i	$Cum.U_i'$	$Cum.U_i$	
	τ_1	10	5	1	0.50	10	0.50	**0**	0	0.5	0.50	
	τ_3	13	6	3	0.46	10	0.60	**3**	0.14	~~1.1~~	~~0.96~~	
	τ_2	25	6	4	0.24	20	0.30	**5**	0.06	0.8	0.74	
	τ_4	25	7	4	0.28	20	0.35	**5**	0.07	1.15	1.02	
2	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	$\Delta\mathbf{T}_i$	ΔU_i	$Cum.U_i'$	$Cum.U_i$	
	τ_3	13	6	3	0.46	13	0.46	**0**	0	0.47	0.46	
	τ_1	10	5	1	0.50	6.5	0.77	**3.5**	0.27	~~1.23~~	~~0.96~~	
	τ_2	25	6	4	0.24	13	0.46	**12**	0.22	0.92	0.70	
	τ_4	25	7	4	0.28	13	0.54	**12**	0.26	1.46	0.98	
3	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	$\Delta\mathbf{T}_i$	ΔU_i	$Cum.U_i'$	$Cum.U_i$	
	τ_2	25	6	4	0.24	25	0.24	**0**	0	0.24	0.24	✓
	τ_4	25	7	4	0.28	25	0.28	**0**	0	0.52	0.52	✓
	τ_3	13	6	3	0.46	12.5	0.48	**0.50**	0.02	1	0.98	✓
	τ_1	10	5	1	0.50	6.26	0.80	**3.75**	0.30	1.32	1.02	
4	τ_i	T_i	C_i	m_i	U_i	T_i'	U_i'	$\Delta\mathbf{T}_i$	ΔU_i	$Cum.U_i'$	$Cum.U_i$	
	τ_2	25	6	4	0.24	25	0.24	**0**	0	0.24	0.24	
	τ_4	25	7	4	0.28	25	0.28	**0**	0	0.52	0.52	
	τ_3	13	6	3	0.46	12.5	0.48	**0.50**	0.02	1	0.98	
	τ_1	10	5	1	0.50	6.25	0.80	**3.75**	0.30	1.32	1.02	

toolset [20]. The SPEC CPU2000 benchmark suite is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers. The benchmarks in this suite were chosen to represent real-world applications, and thus exhibit a wide range of runtime behaviors.

In order to test our algorithm, we generated a group of synthetic task sets. Each of the 26 SPEC CPU2000 benchmarks forms a curve with different points [memory size, execution time]. An exponential-fit model (with the form of $a = \exp(b)$) can thus be obtained with the 95% confidence interval values for a and b for each benchmark.

In our simulations, synthetic task sets were generated by randomly choosing a specific number of tasks n, where each task corresponds to a curve generated from the exponential-fit model of one of the 26 SPEC CPU2000 benchmarks. A thousand task sets are generated for each n. Besides, each time a new curve for a task set was generated, we used random values for a and b that fall into the 95% confidence interval of each of the two parameters.

B. Target Architecture

For the architecture in our experiments, we assume it contains a total of four cores and one cache memory, which is accessible to all cores. Similar architectures can be found commercially [21], [22]. Our cache allocation scheme may be implemented with any cache management scheme that can provide a fixed size of cache unit, and enforce strict isolation guarantees. The implementation is independent of the associativity or the replacement policy, as long as the relationship between execution times and number of cache units are given.

C. Simulation results of testing HBCA1 and HBCA2 approaches

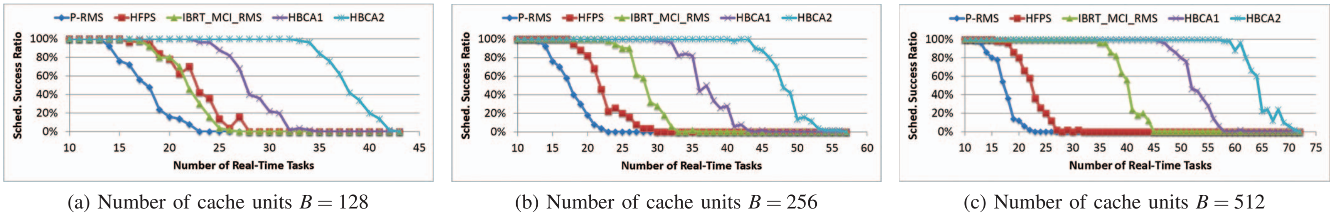
We compare two approaches, i.e. HBCA1 and HBCA2, with three different representative scheduling schemes. The first one is the Partitioned Rate Monotonic Scheduling (P-RMS) algorithm. This is one of the most commonly used approaches for partitioned scheduling on multi-core. A drawback for P-RMS is that it does not take the task period and execution time relationship into consideration for cache allocation and task partitioning. We use this approach as our base line approach. The second approach we investigate is the Harmonic-Fit Fixed-Priority Scheduling (HFPS) algorithm, proposed by Fan et al. [23]. This scheme takes period relationship among multiple tasks into consideration when scheduling fixed-priority tasks on multi-core platforms. Both P-RMS and HFPS do not take the variable execution times with cache allocations into consideration. Therefore, we have to use the WCET values corresponding to the worst-case scenario when $m_i = 1$. The third approach is IBRT-MCI-RMS [16] as mentioned before, which determines cache allocation based on the metric that optimizes the resource usage for a single task. These three scheduling algorithms with both HBCA1 and HBCA2 were employed to schedule the task sets on the architecture discussed above.

We define Schedulability Success Ratio (SSR) as the ratio between the number of successfully scheduled task sets divided by the total task sets tested. Figures 1 and 2 report the SSR for the tested task sets with different number of real-time tasks. Figure 1 shows results using a cache unit size of 1 KB. Figure 2 shows results using a cache unit size of 4 KB.

In Figure 1(a), when task number is around 14 for the case of P-RMS and 20 for the case of HFPS, we can see that the SSR starts decreasing. Also, as the number of cache units increases, as shown in Figure 1(b) and 1(c), we can see that the SSRs of P-RMS and HFPS remain almost constant. This is because they are not memory aware and therefore cannot take advantage of the increase of the number of memory cache units. On the other hand, the methods IBRT_MCI_RMS, HBCA1 and HBCA2 take advantage of the increase of the number of cache units. For instance, the schedulability success ratio of HBCA2 starts decreasing when task number is around 45 in Figure 1(b) (with 256 cache units) and around 60 in Figure 1(c) (with 512 cache units).

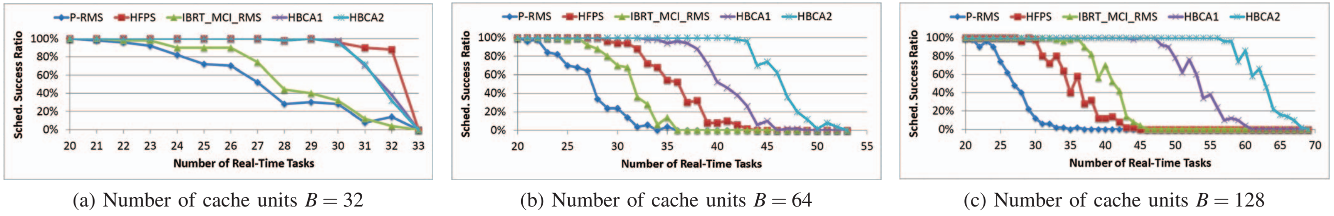
In Figure 2(a), when the task number is around 25 for cases P-RMS, IBRT_MCI_RMS and HFPS, we can see that the SSRs start decreasing. When the task number is around 30, the SSRs start decreasing for HBCA1 and HBCA2. As the number of cache units increases (Figures 2(b) and 2(c)), IBR_MCI_RMS, HBCA1 and HBCA2 starts decreasing their SSR, for example, with task number values around 37, 49 and 58, respectively (see Figure 2(c)).

From the above-mentioned observations, it can be inferred that with a larger cache memory size, the memory-aware mechanisms, and especially our two approaches, are able to schedule a larger number of tasks in the system. One exception to the pattern is Figure 2(a). Note that this is because the number of cache units in this configuration is not large enough for the memory-aware methods to reduce the WCET values in order to increase the number of tasks schedulable in the system.



(a) Number of cache units $B = 128$ (b) Number of cache units $B = 256$ (c) Number of cache units $B = 512$

Figure 1: Number of Tasks VS. Scheduling Success Ratio. Cache Unit Size = 1 KB



(a) Number of cache units $B = 32$ (b) Number of cache units $B = 64$ (c) Number of cache units $B = 128$

Figure 2: Number of Tasks VS. Scheduling Success Ratio. Cache Unit Size = 4 KB

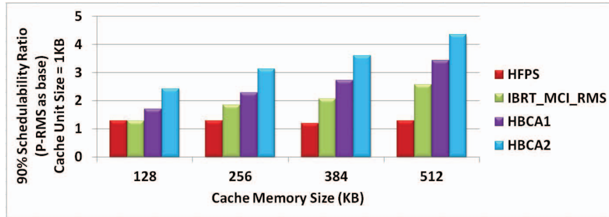


Figure 3: 90% Schedulability Ratio. Cache Unit Size = 1 KB

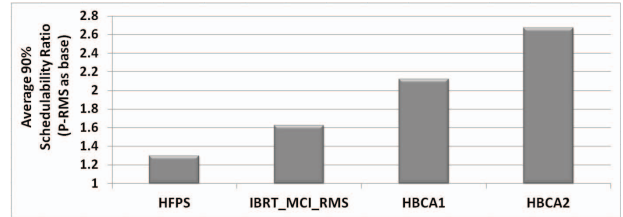


Figure 4: Average 90% Schedulability Ratio

Figure 3 shows the schedulability of each tested mechanism with cache unit size of 1KB. Each mechanism displays the value S (maximum number of tasks such that the SSR of the evaluated method is greater than or equal to 90%) normalized against the S value obtained with P-RMS. For instance, in Figure 1(c), the S values for HBCA1 and HBCA2 are 47 and 62, respectively. It can be seen that HFPS always shows the same improvement, because it is a non-memory-aware mechanism. The remaining mechanisms that are memory-aware show an increasing improvement with the increment of memory cache units available per cache memory. The HBCA2 approach is able to schedule up to 4.1 times more tasks when compared to P-RMS.

Figure 4 shows the average values of S for data using both cache unit sizes (i.e. 1KB and 4KB) and the four cache memory sizes. From the figure, HBCA2 is able to schedule up to 267% more real-time tasks than the P-RMS, and 101%, 64% and 26% more tasks when compared to HFPS, IBRT-MCI-RMS and HBCA1, respectively.

VI. Related Work

As multi-core platforms become more and more pervasive in computing system design, how to manage shared fast local memory, such as cache or scratch-pad memory, in multi-core architectures to improve the predicability and schedulability of real-time applications has attracted more and more research efforts. Cache partitioning has shown to be one of the most effective methods for managing the shared fast local memory while optimizing other design objectives, such as performance maximization [24], quality-of-service (QoS) enhancement [25], and fairness [6]. Existing work on cache partitioning can be largely categorized into two groups [26]: *cache allocation policies* and *cache management schemes*. The first ones focus on policies to dictate how to allocate available cache resources to different tasks to achieve different objectives, such as fairness, priorities, and performance maximization (e.g. [8], [27], [28]). The second ones intend to enforce, by means of hardware or software, the distribution of the outcomes of the cache allocations so that each program can access its allocated cache memory (e.g. [5], [9], [29], [30]).

We are interested in developing static cache allocation policies for real-time systems to enhance the predicability and schedulability when scheduled in a multi-core environment. Unlike our proposed allocation policies, some techniques have been proposed for single-core platforms [31], [32], and some others use a non-preemptive EDF policy for intra-core scheduling [33], [34]. A few approaches that have been published are closely related to our work. Chang et. al. [16], [35] develop a series of algorithms for real-time systems scheduled based on EDF in island-based multi-core real-time systems with local and global heterogeneous memories. The algorithm, so called

Island Based Real-Time Scheduling for Multi-Core Islands (IBRT-MCI), intends to optimize the system resource (CPU and fast local memory) usage for a single task. A variant of this algorithm is also introduced in a later publication [16], in which the intra-core scheduling is performed according to RMS, i.e. IBRT-MCI-RMS. As discussed before, the optimal solution that can optimize the system resource usage for a single task does not necessarily optimize that for the entire task set. Also, as we show in our simulation results, incorporating period relation into cache allocation and task mapping can significantly improve the schedulability of real-time systems. Kim et. al. [36] propose a cache allocation policy that relies on page coloring as the cache management scheme. Different from our approach, their algorithm assigns cache units privately to cores instead of tasks, thus allowing intra-core cache units sharing. This alleviates the memory co-partitioning problem due to the page coloring management scheme, but increases the predictability analysis complexity. Suzuki et. al. [7] propose two algorithms as cache allocation policies, taking into consideration the cache memory partitions and the main memory banks assigned to each task. Unlike our approaches, such algorithms assume EDF as intra-core scheduling policy instead of RMS.

VII. Conclusions

We study the cache allocation and task partitioning problem when running a set of fixed-priority real-time tasks on a multi-core platform sharing a common cache memory. We have developed two static schemes for cache allocation and task partitioning. The first one (HBCA1) combines two previous research studies that take task variable WCET times and period relationship into consideration. The second one (HBCA2) is a more elaborate approach that can judiciously choose the cache size for each task, while exploiting the task harmonic relationships within the task set. Our simulation results show that our second approach increases the schedulability of real-time tasks up to four times, when compared to a conventional Partitioned Rate Monotonic Scheduling (P-RMS). As future work, we plan to extend this research to analyze different types and levels of memories, within the memory hierarchy, using the approaches proposed in this work.

References

[1] L. A. D. Bathen and N. D. Dutt, "Software Controlled Memories for Scalable Many-Core Architectures," *IEEE RTCSA*, 2012.

[2] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling, "Multicore in real-time systems–temporal isolation challenges due to shared resources," *WICERT, IEEE DATE*, 2013.

[3] M. Taylor, "A landscape of the new dark silicon design regime," *IEEE MICRO*, 2013.

[4] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-Aware Scheduling and Analysis for Multicores," *ACM EMSOFT*, 2009.

[5] B. Lesage, I. Puaut, and A. Seznec, "PRETI : Partitioned REal-TIme shared cache for mixed-criticality real-time systems," *ACM RTNS*, 2012.

[6] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan, "Gaining Insights into Multicore Cache Partitioning : Bridging the Gap between Simulation and Real Systems," *IEEE HPCA*, 2008.

[7] N. Suzuki, H. Kim, D. D. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar, "Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses," *IEEE CSE*, 2013.

[8] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," *IEEE RTAS*, 2013.

[9] G. Gracioli and A. Fröhlich, "An Experimental Evaluation of the Cache Partitioning Impact on Multicore Real-Time Schedulers," *IEEE RTCSA*, 2013.

[10] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys*, no. 43.4, 2011.

[11] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstr, "The Worst-Case Execution Time Problem Overview of Methods and Survey of Tools," *ACM TECS*, 2008.

[12] C. Chang, J. Chen, T. Kuo, and H. Falk, "Real-Time Partitioned Scheduling on Multi-Core Systems with Local and Global Memories," in *ASP-DAC*, 2013.

[13] F. Eisenbrand and N. Hähnle, "Scheduling periodic tasks in a hard real-time environment," *Automata, languages and programming. Springer Berlin Heidelberg*, 2010.

[14] M. Fan and G. Quan, "Harmonic-Aware Multi-Core Scheduling For Fixed-Priority Real-Time Systems," *IEEE TPDS*, 2014.

[15] J. Cantin and M. Hill, "Cache performance for selected SPEC CPU2000 benchmarks," *ACM SIGARCH Comp. Arch. News*, no. 29.4, 2001.

[16] C.-W. Chang, J.-J. Chen, T.-W. Kuo, and H. Falk, "Real-Time Task Scheduling on Island-Based Multi-Core Platforms," *IEEE TPDS*, 2015.

[17] J. W. Liu, *Real-time systems*. Prentice Hall PTR, 2000.

[18] C. Han and H. Tyan, "A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms," *IEEE RTSS*, 1997.

[19] J. Henning, "SPEC CPU2000: Measuring CPU performance in the new millennium," *Computer*, vol. 33.7, 2000.

[20] D. Burger and T. Austin, "The SimpleScalar tool set, version 2.0," *ACM SIGARCH Comp. Arch. News*, no. 25.3, 1997.

[21] Y. Zhang, L. Peng, B. Li, J. K. Peir, and J. Chen, "Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications," *IEEE IISWC*, 2011.

[22] Y. Iwase, D. Abe, and T. Yakoh, "GPGPU aided method for real-time systems," *IEEE INDIN*, 2012.

[23] M. Fan and G. Quan, "Harmonic-Fit Partitioned Scheduling for Fixed-Priority Real-Time Tasks on the Multiprocessor Platform," *IEEE/IFIP EUC*, 2011.

[24] L. Hsu, S. Reinhardt, R. Iyer, and S. Makineni, "Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource," *ACM PACT*, 2006.

[25] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS policies and architecture for cache/memory in CMP platforms," *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[26] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," *ACM SIGARCH Comp. Arch. News*, no. 39.3, 2011.

[27] M. K. Qureshi and Y. N. Patt, "Utility-Based Cache Partitioning : A Low-Overhead , High-Performance , Runtime Mechanism to Partition Shared Caches," *IEEE/ACM MICRO*, 2006.

[28] D. Tam, R. Azimi, L. Soares, and M. Stumm, "Managing Shared L2 Caches on Multicore Systems in Software," *IEEE WIOSCA-ISCA*, 2007.

[29] V. Suhendra and T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores," *ACM DAC*, 2008.

[30] B. Ward, J. Herman, C. Kenna, and J. Anderson, "Making Shared Caches More Predictable on Multicore Platforms," *IEEE ECRTS*, 2013.

[31] B. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real-time embedded systems," *IEEE RTCSA*, 2008.

[32] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "Evaluation of Cache Partitioning for Hard Real-Time Systems," *IEEE ECRTS*, 2014.

[33] B. Berna and I. Puaut, "{PDPA}: Period Driven Task and Cache Partitioning Algorithm for Multi-Core Systems," *ACM RTNS*, 2012.

[34] M. Paolieri, E. Quiñones, F. J. Cazorla, R. I. Davis, and M. Valero, "IA3: An interference aware allocation algorithm for multicore hard real-time systems," *IEEE RTAS*, 2011.

[35] C.-W. Chang, J.-J. Chen, T.-W. Kuo, and H. Falk, "Real-time partitioned scheduling on multi-core systems with local and global memories," *IEEE ASP-DAC*, 2013.

[36] H. Kim, A. Kandhalu, and R. Rajkumar, "A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems," *IEEE ECRTS*, 2013.