A Coordinated Approach for Practical OS-Level Cache Management in Multi-Core Real-Time Systems

Hyoseung Kim, Arvind Kandhalu, Ragunathan (Raj) Rajkumar
Carnegie Mellon University
{hyoseung, arvindkr}@cmu.edu, raj@ece.cmu.edu

***Abstract*—Many modern multi-core processors sport a large shared cache with the primary goal of enhancing the statistic performance of computing workloads. However, due to resulting cache interference among tasks, the uncontrolled use of such a shared cache can significantly hamper the predictability and analyzability of multi-core real-time systems. Software cache partitioning has been considered as an attractive approach to address this issue because it does not require any hardware support beyond that available on many modern processors. However, the state-of-the-art software cache partitioning techniques face two challenges: (1) the memory co-partitioning problem, which results in page swapping or waste of memory, and (2) the availability of a limited number of cache partitions, which causes degraded performance. These are major impediments to the practical adoption of software cache partitioning.**

In this paper, we propose a practical OS-level cache management scheme for multi-core real-time systems. Our scheme provides predictable cache performance, addresses the aforementioned problems of existing software cache partitioning, and efficiently allocates cache partitions to schedule a given taskset. We have implemented and evaluated our scheme in Linux/RK running on the Intel Core i7 quad-core processor. Experimental results indicate that, compared to the traditional approaches, our scheme is up to 39% more memory space efficient and consumes up to 25% less cache partitions while maintaining cache predictability. Our scheme also yields a significant utilization benefit that increases with the number of tasks.

I. Introduction

Modern multi-core processors incorporate shared resources among cores to improve performance and efficiency. Among them, an on-chip large shared cache has received much attention [12][13][33]. The shared cache can efficiently bridge the performance gap between memory and processor speeds by backing up small private caches. Each of the cores can access the entire shared cache, so a better cache hit ratio can be statistically achieved. Due to these benefits, the size of the shared cache has become increasingly larger. For example, the Intel Core i7 has 8MB of a shared L3 cache, and the ARM Cortex A15 architecture can have up to 4MB of a shared L2 cache.

While the use of a shared cache can reduce the average execution time of a task, it introduces significant worst-case timing penalties due to "cache interference". Cache interference in multi-core systems can be categorized into two types: *inter-core* and *intra-core*. Inter-core cache interference happens when tasks running on different cores access the shared cache simultaneously. Since the execution of a task may be potentially affected by memory accesses of *all* tasks running on other cores, the accurate analysis of inter-core cache interference is extremely difficult [13]. Intra-core cache interference, in contrast, occurs within a core. When a task preempts another task, the preempting task may evict the cache contents of the preempted task. Moreover, while a task is inactive, other tasks can corrupt its cache. It has been shown in [15] that inter-core and intra-core cache interference on a state-of-the-art quad-core processor increased the task completion time by up to 40% and 27%, respectively, compared to when it runs alone in the system. As the number of cores and the size of the shared cache increases, the negative impact of cache interference becomes more significant.

Many researchers in the real-time systems community have recognized and studied the problem of cache interference in order to use the shared cache in a predictable manner. Among a variety of approaches, software cache partitioning, called *page coloring*, has been considered as an appealing approach to address this issue. Page coloring prevents cache disruptions from other tasks by assigning exclusive cache partitions to each task. It does not require any hardware support beyond that available on most of today's multi-core processors.

There still remain two challenging problems to be solved before page coloring can be used widely in multi-core real-time systems. The first problem is the memory co-partitioning problem [20][21]. Page coloring simultaneously partitions the entire physical memory into the number of cache partitions. If a certain number of cache partitions is assigned to a task, the same number of memory partitions is also assigned to that task. However, a task's memory usage is not necessarily related to its cache usage. If a task requires more number of memory partitions than that of cache partitions, the required memory partitions should be assigned to the task despite its small cache usage. Otherwise, the task would suffer from page swapping. If a task requires more number of cache partitions than that of memory partitions, some of the assigned memory would be wasted. We are not aware of any previous work that has provided a software-level solution for this problem.

The second problem is the availability of a limited number of cache partitions. As the number of tasks increases, the amount of cache that can be used for an individual task becomes smaller and smaller resulting in degraded performance. Moreover, the number of cache partitions may not be enough for each task to have its own cache partition. This second problem also unfortunately applies to hardware-based cache partitioning schemes.

In this paper, we propose a practical OS-level cache management scheme for a multi-core real-time system that uses partitioned fixed-priority preemptive scheduling. Our scheme provides predictable cache performance and addresses the aforementioned problems of page coloring through tight coordination of *cache reservation*, *cache sharing*, and *cache-aware task allocation*. Cache reservation ensures the exclusive use of a certain amount of cache for individual cores to prevent inter-core cache interference. Within each core, cache sharing allows tasks to share the reserved cache, while providing a safe upper bound on intra-core cache interference. Cache sharing also significantly mitigates the memory co-partitioning problem and the limitations on the number of cache partitions. By using cache reservation and cache sharing, cache-aware task allocation determines efficient task and cache allocation to schedule a given taskset.

Our scheme does *not* require special hardware cache partitioning support or modifications to application software. Hence, it is readily applicable to commodity processors such as the Intel Core i7. Our scheme can be used not only for developing a new system but also for migrating existing applications from single-core to multi-core platforms.

Contributions: Our contributions are as follows:

- We introduce the concept of sharing cache partitions under page coloring to counter the memory co-partitioning problem and the limited number of cache partitions. We show how pages are allocated when cache partitions are shared, and provide a condition that checks the feasibility of sharing while guaranteeing the allocation of the required memory to tasks.
- We provide a response time test for checking task schedulability when cache partitions are shared among tasks. Our approach is independent of the specific cache analysis used and allows estimating the worst-case execution time (WCET) of a task in isolation from other tasks.
- Our cache-aware task allocation algorithm reduces the number of cache partitions required to schedule a given taskset, while meeting both the task memory requirements and the task timing constraints. We also show that the remaining cache partitions after the allocation can be used to save the total CPU utilization.
- We have implemented and evaluated our scheme by extending the Linux/RK platform [25][29] running on the Intel Core i7 quad-core processor.[1] The experimental results on a real machine demonstrate the effectiveness of our scheme.

Organization: The rest of this paper is organized as follows. Section II reviews related work and describes the assumptions and notation used in this paper. Section III presents our coordinated cache management scheme. A detailed evaluation of our scheme is provided in Section IV. Finally, we conclude the paper in Section V.

II. Related Work and Background

We discuss related work on cache interference and describe the assumptions and notation used in our paper.

A. Related Work

Hardware cache partitioning is a technique for avoiding cache interference by allocating private cache space to each task in a system. With cache partitioning, the system performance is largely dependent on how cache partitions are allocated to tasks. Yoon et al. [33] formulated cache allocation as a MILP problem to minimize the total CPU utilization of Paolieri's new multi-core architecture [26]. Fu et al. [12] proposed a sophisticated low-power scheme that uses both cache partitioning and DVFS. In [27], the authors focused on a system using non-preemptive partitioned scheduling and proposed a task allocation algorithm that also allocates cache partitions. These approaches, however, assume special underlying hardware cache partitioning support, which is not yet widely available in current commodity processors [1][3][31].

Software-based page coloring is an alternative to hardware cache partitioning support. Wolfe [32] and Liedtke et al. [20] used page coloring to prevent cache interference in a single-core real-time system. Bui et al. [8] focused on improving the schedulability of a single-core system with page coloring. Page coloring also has been studied for multi-core systems in [10][30]. Guan et al. [13] proposed a non-preemptive scheduling algorithm for a multi-core real-time system using page coloring. Lin et al. [21] evaluated existing hardware-based cache partitioning schemes on a real machine via page coloring. Unfortunately, these approaches do not provide a software method to tackle the problems of memory co-partitioning and the limited cache partitions. Zhang et al. [34] proposed a *hot-page* coloring approach that assigns cache partitions only to a small set of frequently accessed pages. However, since they use on-line page access monitoring and page migration, it may not be suitable for time-critical systems.

For single-core real-time systems, much research has been conducted on the analysis of cache interference penalties caused by preemptions. The cache penalties are bounded by accounting them as cache-related preemption delays while performing schedulability analysis. Altmeyer et al. [4], Lunniss et al. [23] and Lee et al. [18] focused on reducing the cache penalties by using static cache analyses. However, they do not consider cache partitioning that can prevent the cache penalties by assigning exclusive cache partitions to tasks. Our scheme is independent of the specific cache analysis used and allows both sharing and exclusive use of cache partitions. Busquets-Mataix et al. [9] proposed a hybrid technique of cache partitioning and schedulability analysis for a single core system, but it cannot be directly applied to a shared cache of a multi-core processor.

Pellizzoni et al. [28] suggested a compiler-assisted approach for cache predictability. Based on source-level user annotations, their proposed compiler divides a program into small blocks, each of which is non-preemptively scheduled and prefetches all its required data into the cache before execution. This approach can provide predictability on a private cache but not on a shared cache.

B. Page Coloring

We briefly describe the background on the page coloring technique, on which our scheme is based. The key to the

[1] Intel Core i7 processors are used in not only desktop/server machines but also aerospace and defense embedded systems [2]. In fact, Intel's Embedded Systems Division is rather big inside Intel.

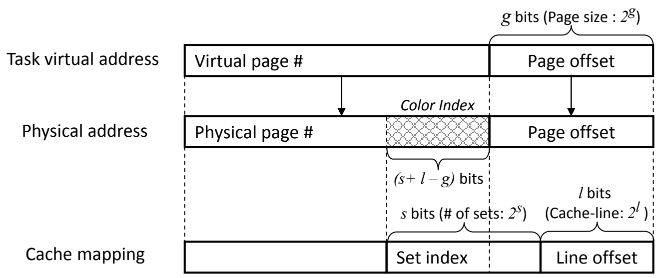


Fig. 1. Memory to cache mapping and page coloring

page coloring technique lies in the mapping between cache entries and physical addresses. Figure 1 shows how a task's memory address is mapped to a cache entry. With paged virtual memory, every memory address referenced by a task represents a virtual address in which the g least significant bits are used as an offset into a page, and the remaining bits of the virtual address are translated into a physical page number. The cache location to store memory contents is identified by the physical address. We assume a cache memory with 2^l bytes per cache-line and 2^s cache sets in this figure. Then, the last l bits of the physical address are used as a cache-line offset, and the preceding s bits are used as a set index into the cache. As can be seen, there are overlapping intersection bits between the physical page number and the set index. Page coloring uses these intersection bits as a *color index* which partitions the cache into $2^{(s+l-g)}$ cache partitions. Simultaneously, the color index co-partitions the entire physical memory into $2^{(s+l-g)}$ memory partitions. In other words, physical memory pages with the same color index are also grouped into a memory partition, and each memory partition corresponds to a cache partition with the same color index. Since the OS can control the physical pages and the virtual↔physical address translation, specific cache partitions can be assigned to a task by allocating memory pages in the corresponding memory partitions to that task.

C. Assumptions and Notation

We consider a system equipped with a single-chip multi-core processor and M_{total} MB of memory. The processor has N_C identical cores running at a fixed clock speed and a unified last-level cache shared among all the cores.[2] We adopt page coloring to manage the shared cache in OS software. With page coloring, the cache is divided into N_P partitions. Each cache partition is represented as a unique integer in the range from 1 to N_P. The entire memory is also divided into N_P memory partitions of M_{total}/N_P MB.

We focus on systems that use *partitioned fixed-priority preemptive task scheduling*. Tasks are ordered in the decreasing order of priorities, i.e. $i < j$ implies that task τ_i has higher priority than task τ_j. We assume that each task has a unique priority and n is the lowest priority. Task τ_i is represented as follows:

$$\tau_i = \{C_i^p, T_i, D_i, M_i\}$$

- C_i^p: the worst-case execution time of task τ_i, when it runs alone in a system with p cache partitions assigned to it. We have, $\lceil \frac{M_i}{M_{total}/N_P} \rceil \leq p \leq N_P$
- T_i: the period of τ_i
- D_i: the relative deadline of τ_i ($D_i \leq T_i$)
- M_i: the size of required physical memory in MB, which should be assigned to τ_i to prevent swapping.

The minimum p for C_i^p depends on M_i due to the memory co-partitioning that page coloring causes. The possible C_i^p values of task τ_i are assumed to be known ahead of time. We specifically consider measurement-based WCET estimates, where the C_i^p values can be measured by changing the number of cache partitions allocated to τ_i. We do not assume the use of a static cache analysis tool since it may not be available for a target processor's unified shared cache. However, it must be noted that advances in static cache analysis can benefit from our approach (and vice-versa). Section IV-B will describe more details on how we estimate the WCET. We assume that C_i^p is non-increasing with p,[3] i.e. $p < p' \implies C_i^p \geq C_i^{p'}$. In the rest of the paper, C_i may be used as a simplified representation of C_i^p, when task τ_i's p is obvious, or when each task is assumed to be assigned its own p.

We also use the following notation for convenience:

- $hp(i)$: the set of tasks with higher priorities than i
- $hep(i)$: the set of tasks whose priorities are higher than or equal to i
- $int(j, i)$: the set of tasks whose priorities are lower than j and higher than or equal to i

It is assumed that a task does not suspend itself during its execution. For simplicity, we further assume that tasks do not share memory. For a description of how our scheme can be used for tasks that use shared memory segments, please see [15].

III. Coordinated Cache Management

In this section, we describe our proposed cache management scheme. Figure 2 shows the overview of our scheme that consists of three components: *cache reservation*, *cache sharing*, and *cache-aware task allocation*. Cache reservation ensures the exclusive use of a portion of the shared cache for each core. Cache sharing enables sharing of cache partitions among tasks within each core. Cache-aware task allocation uses these two components to find efficient cache and task allocation while maintaining feasibility.

A. Cache Reservation

Due to the inherent difficulties of precisely analyzing inter-core cache interference on a multi-core processor, we reserve a portion of cache partitions for each core to prevent inter-core cache interference. The reserved cache partitions are exclusively used by their owner core, thereby preventing cache contention from other cores. Per-core cache reservation differentiates our scheme from other cache partitioning techniques that allocate exclusive cache partitions to each task. Within each core, cache partitions reserved for the core *can* be

[2]This corresponds to various modern multi-core processors, such as Intel Core i7, AMD FX, ARM Cortex A15, and FreeScale QorIQ processors. In contrast, tiled multi-cores, such as Tilera TILE64, typically do not have a shared cache, but we focus on the former type of architecture in this work.

[3]Since C_i^p is non-increasing in p, it can begin to plateau at some point. At this point, adding more cache partitions will not reduce a task's execution time.

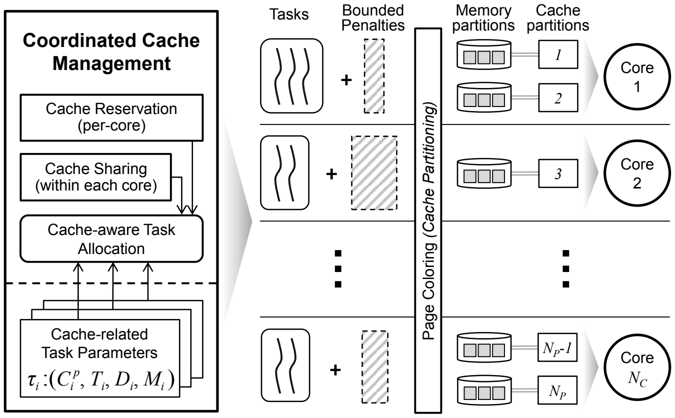


Fig. 2. Overview of the proposed OS-level cache management

shared by tasks running on the core. This approach allows the core to execute more tasks than the number of cache partitions allocated to that core. The execution time of a task can potentially be further reduced by providing more cache partitions to the task. Moreover, since the sharing of a cache partition means the sharing of an associated memory partition, it can significantly reduce the waste of cache and memory resources caused by the memory co-partitioning problem due to page coloring.

Cache partitions are reserved for a core by allocating associated memory partitions to the core. Each core manages the state of pages in its memory partitions. When a new task is assigned to a core, the task's memory requests are handled by allocating free pages from the core's memory partitions. The appropriate number of cache partitions for each core depends on the tasks running on the core. This cache allocation will be determined by our cache-aware task allocation, discussed in Section III-D.

B. Cache Sharing: Bounding Intra-core Penalties

Suppose that a certain number of cache partitions is allocated to a core by cache reservation. Our scheme allows tasks running on the core to share the given partitions, but sharing causes intra-core cache interference. Intra-core cache interference can be further subdivided into two types:

1) *Cache warm-up delay*: occurs at the beginning of each period of a task and arises due to the execution of other tasks while the task is inactive.
2) *Cache-related preemption delay*: occurs when a task is preempted by a higher-priority task and is imposed on the preempted task.

Previous work on bounding cache interference on single-core platforms [4][18][23] assumes that the cache warm-up delay can be taken into account in the WCET of a task by static cache analyses. However, such static cache analysis tools may not be readily available for modern multi-core processors. We therefore consider the cache warm-up delay as an extrinsic factor to a task's WCET. This approach enables measurement-based WCET analysis to estimate the task WCET in isolation from other tasks.[4] For instance, once a task is launched, the task's cache is initially warmed up during the startup phase or the very first execution of the task. If the task runs alone in the system or uses its cache all by itself, subsequent task instances do not experience any cache warm-up delay at run-time [20]. By considering the cache warm-up delay as an extrinsic factor, the WCET obtained in such an isolated environment can be safely used even when the task's cache is shared.

We formally define cache warm-up delay and cache-related preemption delay. $\omega_{j,i}$ is τ_j's cache warm-up delay, which is caused by the tasks belonging to $hep(i)$ and sharing cache partitions with τ_j. $\gamma_{j,i}$ is the cache-related preemption delay caused by τ_j and imposed on the tasks that belong to $int(j,i)$ and share cache partitions with τ_j. Hence, $\omega_{j,i}$ and $\gamma_{j,i}$ are represented as follows:

$$\omega_{j,i} = \left| S(j) \cap \bigcup_{k:k\neq j \wedge k\in hep(i)} S(k) \right| \cdot \Delta$$

$$\gamma_{j,i} = \left| S(j) \cap \bigcup_{k\in int(j,i)} S(k) \right| \cdot \Delta$$

- $S(j)$: the set of cache partitions assigned to τ_j
- Δ : the time to refill one cache partition, which is constant and architecture-dependent.

Each core's utilization with intra-core cache interference penalties, ω and γ, can be calculated by extending Liu and Layland's schedulability condition [22] as follows:

$$U = \sum_{i=1}^{n} \left(\frac{C_i}{T_i} + \frac{\omega_{i,n}}{T_i} + \frac{\gamma_{i,n}}{T_i} \right) \leq n(2^{1/n} - 1) \qquad (1)$$

where U is the total CPU utilization of a core. It is based on the Basumallick and Nilsen's technique [5], but we explicitly consider cache warm-up delay ω.

The iterative response time test [14] can be extended as follows to incorporate the two types of intra-core cache interference:

$$R_i^{k+1} = C_i + \omega_{i,n} + \sum_{j\in hp(i)} \left(\left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \right) + \sum_{j\in hp(i)} \left\{ \omega_{j,n} + \left(\left\lceil \frac{R_i^k}{T_j} \right\rceil - 1 \right) \omega_{j,i} \right\} + \sum_{j\in hp(i)} \left(\left\lceil \frac{R_i^k}{T_j} \right\rceil \gamma_{j,i} \right) \qquad (2)$$

where R_i^k is the worst-case response time of τ_i at the k^{th} iteration. The test terminates when $R_i^{k+1} = R_i^k$. Task τ_i is schedulable if its response time is before its deadline: $R_i^k \leq D_i$. We represent the amount of ω and γ delays caused by the execution of a higher priority task τ_j within the worst-case response time R_i^k in the second and the third summing terms of (2). Note that the first execution of a higher priority task τ_j within R_i^k causes a cache warm-up delay of $\omega_{j,n}$, but the subsequent executions of τ_j cause only $\omega_{j,i}$ because tasks with lower priorities than i are not scheduled while τ_i is running.

Figure 3 shows an example taskset $\{\tau_1, \tau_2, \tau_3\}$ sharing a set of cache partitions $\{1, 2\}$. Assume that the cache partitions are pre-assigned to tasks; $S(1)$ is $\{1, 2\}$; $S(2)$ is $\{1\}$; $S(3)$ is $\{2\}$. All tasks have the same execution time $C_i = 2$ and the same periods and deadlines $T_i = D_i = 12$. The cache partition refill time Δ is 1 in this example. When τ_1 starts its execution, it needs to refill its two cache partitions. τ_2 has one cache warm-

[4] Appropriate "error margins" that are proportional to system criticality can be applied to these measurements, as is done in practice.

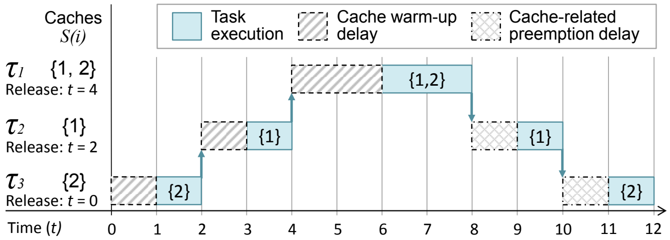


Fig. 3. Three tasks sharing cache partitions with cache penalties

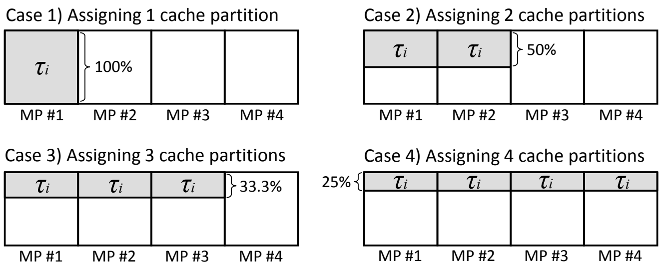


Fig. 4. Page allocations for different cache allocation scenarios

up delay and one cache-related preemption delay due to τ_1. τ_3 also has one cache warm-up delay and one cache-related preemption delay.

It is worth noting that Equations (1) and (2) are independent of the specific cache analysis used. If a precise cache analysis tool is available for a target multi-core processor's shared cache, the cache partition refilling time Δ can be more tightly estimated.

C. Cache Sharing: How to Share Cache Partitions

We now describe how cache partitions are allocated to tasks within a core such that schedulability is preserved and memory requirements are guaranteed despite sharing the partitions. There are two conditions for a cache allocation to be feasible. The first condition is the response time test given by Equation (2). The factors affecting a task's response time are as follows: (i) cache-related task execution time C_i^p, (ii) cache partition refill time Δ, (iii) the number of other tasks sharing the task's cache partitions, and (iv) the periods of the tasks sharing the cache partitions. Factors (i) and (ii) are explicitly used to calculate the response time. If factor (iii) increases or factor (iv) is relatively short, the response time may be lengthened due to cache penalties caused by frequent preemptions.

The second condition is related to the task memory requirements. Before defining this condition, we show in Figure 4 an example of page allocations for different cache allocation cases. In each case, there are four memory partitions and one task τ_i. Each memory partition is depicted as a square and the shaded area represents the memory space allocated to τ_i. The task τ_i's memory requirement M_i is equal to the size of one memory partition. If we assign only one cache partition to τ_i, all pages for τ_i are allocated from one memory partition (Case 1 in Figure 4). If we assign more than one cache partition to τ_i, our scheme allocates pages to τ_i from the corresponding memory partitions in round-robin order.[5] Thus,

[5]If a page is deallocated from τ_i, the deallocated page is used ahead of never-allocated free pages to service τ_i's next page request. This enables multiple memory partitions to be allocated at the same rate without explicit enforcement such as in memory reservation [11].

Algorithm 1 MinCacheAlloc(Γ^j, N_P^j)

Input: Γ^j: a taskset assigned to the core j, N_P^j: the number of available cache partitions in the core j
Output: φ_{min}: a cache allocation with the minimum CPU utilization ($\varphi_{min} = \varnothing$, if no allocation is feasible), $minUtil$: the CPU utilization of Γ^j with φ_{min}

```
φ_min ← ∅; minUtil ← 1
Φ ← a set of candidate allocations of N_P^j to Γ^j
for each allocation φ_i in Φ do
    Apply φ_i to Γ^j
    if Γ^j satisfies both Eq. (2) and Eq. (3) then
        currentUtil ← CPU utilization from Eq. (1)
        if minUtil ≥ currentUtil then
            φ_min ← φ_i; minUtil ← currentUtil
return {φ_min, minUtil}
```

Algorithm 2 FindBestFit(τ_i, N_C, A_T, A_P)

Input: τ_i: a task to be allocated, N_C: the number of cores, A_T: an array of a taskset allocated to each core, A_P: an array of the number of cache partitions assigned to each core
Output: cid: the best-fit core's index ($cid = 0$, if no core can schedule τ_i)

```
space ← 1; cid ← 0
for j ← 1 to N_C do
    {φ, util} ← MinCacheAlloc(τ_i ∪ A_T[j], A_P[j])
    if φ ≠ ∅ and space ≥ 1 − util then
        space ← 1 − util; cid ← j
return cid
```

---

the same amount of pages from each of the corresponding memory partitions is allocated to τ_i at its maximum memory usage (Cases 2, 3, and 4 in Figure 4). The reason behind this approach is to render the page allocation deterministic, which is required for each task's cache access behavior to be consistent. For instance, if pages are allocated randomly, a task may have different cache performance when it re-launches.

A cache partition can be shared among tasks by sharing a memory partition. We present a necessary and sufficient condition for cache sharing to meet the task memory requirements under our page allocation approach. For each cache partition ρ, the following condition must be satisfied:

$$\sum_{\forall \tau_i:\, \rho \in S(i)} \frac{M_i}{|S(i)|} \le M_{total}/N_P \qquad (3)$$

where M_i is the size of the memory requirement of τ_i, $|S(i)|$ is the number of cache partitions assigned to τ_i, and M_{total}/N_P is the size of a memory partition. $\frac{M_i}{|S(i)|}$ represents τ_i's per-memory-partition memory usage. This condition means that the sum of the per-memory-partition usage of the tasks sharing the cache partition ρ should not exceed the size of one memory partition. If this condition is not satisfied, tasks may experience memory pressure or swapping.

Algorithm 1 shows a procedure for finding a feasible cache allocation with the minimum CPU utilization. It first creates a set of candidate cache allocations to be examined, which are combinations of given cache partitions for a given taskset. Then, it checks the feasibility of each candidate allocation by using Equation (2) and (3). Many methods can be used to generate the candidate cache allocations, such as exhaustive

Algorithm 3 CacheAwareTaskAlloc(Γ, N_C, N_P)

Input: Γ: a taskset to be allocated, N_C: the number of cores, N_P: the number of available cache partitions

Output: True/False: the schedulability of Γ, A_T: an array of a taskset allocated to each core, A_P: an array of the number of cache partitions assigned to each core, $N_{P'}$: the number of remaining cache partitions

```
Sort tasks in Γ in decreasing order of their average utilization
Initialize elements of A_T to ∅ and A_P to 0
for each task τ_i in Γ do
    cid ← FindBestFit(τ_i, N_C, A_T, A_P)
    if cid > 0 then                              ▷ Found the core for τ_i
        Insert τ_i to A_T[cid]
        Mark τ_i schedulable
        continue
    for k ← 1 to N_P do                          ▷ Try with k more partitions
        for j ← 1 to N_C do
            A_tmp[j] ← A_P[j] + k
        cid ← FindBestFit(τ_i, N_C, A_T, A_tmp)
        if cid > 0 then
            Insert τ_i to A_T[cid]
            Mark τ_i schedulable
            N_P ← N_P − k                        ▷ Assign k to the core
            A_P[cid] ← A_P[cid] + k
            break
if all tasks schedulable then
    return {True, A_T, A_P, N_P}
else
    return {False, A_T, A_P, N_P}
```

search and heuristics. An efficient way to generate candidate allocations is part of our future work.

D. Cache-Aware Task Allocation

Cache-aware task allocation is an algorithm to allocate tasks and cache partitions to cores while exploiting the benefits of cache reservation and cache sharing. The objective of our algorithm is to reduce the number of cache partitions required to schedule a given taskset on a given number of cores, because remaining cache partitions can be used for many purposes, such as for non-real-time tasks or for saving the CPU utilization.

Under our scheme, tasks may share cache partitions when they are assigned to the same core. This means that, to take advantage of cache sharing, it is desired to pack tasks into the same core as much as possible. Hence, our algorithm is based on the best-fit decreasing bin-packing algorithm that results in load concentration. For cache allocation, our algorithm gradually assigns cache partitions to cores while allocating tasks to cores by using cache reservation and cache sharing.

We first explain Algorithm 2 that finds the best-fit core in our task allocation algorithm. Once the task to be allocated is given, Algorithm 2 checks whether the task is schedulable on each core and estimates the total utilization of each core with the task. Then, it selects the core where the task fits best.

Our cache-aware task allocation algorithm is given in Algorithm 3. Before allocating tasks, it sorts tasks in decreasing order of their average utilization, i.e. $(\sum_{p=1}^{N_P} C_i^p / N_P)/T_i$. The number of cache partitions for each core is set to zero. Then, the algorithm initiates task allocation. If a task to be allocated is not schedulable on any core and the number of remaining cache partitions is not zero, the algorithm increases the number of each core's cache partitions by 1 and finds the best-fit core again, until the cache partition increment per core exceeds N_P. When the algorithm finds the best-fit core, only the best-fit core maintains its increased number of cache partitions and other cores return to their previous number of cache partitions.

The algorithm returns the number of remaining cache partitions along with the task allocation and cache assignment. Here, we employ a simple solution to save the CPU utilization with the remaining cache partitions: assigning each remaining cache partition to a core which will obtain the greatest saving in utilization when an additional cache partition is given to it. We use this approach in our experiments when we measure the CPU utilization with a specified number of cache partitions.

IV. Evaluation

In this section, we evaluate our proposed cache management scheme. We first describe the implementation of our scheme and then show the experimental results of cache reservation, cache sharing, and cache-aware task allocation.

A. Implementation

We have implemented our scheme in Linux/RK, based on the Linux 2.6.38.8 kernel. To easily implement page coloring, we have used the memory reservation mechanism [11][16] of Linux/RK. Memory reservation maintains a global page pool to manage unallocated physical pages. In this page pool, we categorize pages into memory partitions with their color indices. When a real-time taskset is given, our scheme assigns a core index and color indices to each task. Then, a memory reservation is created for each task from the page pool, using the task's memory demand and assigned color indices, and each task only uses pages within its memory reservation during execution.

The target system is equipped with the Intel Core i7-2600 3.4GHz quad-core processor. The system is configured for 4KB page frames and a 1GB memory reservation page pool. The processor has a unified 8MB L3 shared cache that consists of four cache slices. Each cache slice has 2MB and is 16-way set associative with a line size of 64B, thereby having 2048 sets. For the entire L3 cache to be shared among all cores, the processor distributes all physical addresses across the four cache slices by using an on-chip hash function [3][19].[6] Figure 5 shows the implementation of page coloring on this cache configuration. Regardless of the hash function, the cache set index for a given physical address is independent from the cache slice index. Hence, with page coloring, we can use $2^{11+6-12} = 32$ colors and each cache partition spans the four cache slices. Page coloring divides the L3 cache into 32 cache partitions of 256KB and the page pool into 32 memory partitions of 32MB. The cache partition refill time Δ in the target system is 45.3 μsec,[7] which is an empirically obtained from a cache calibration tool, as given in [24].

[6] Intel refers to this technique, which is unrelated to cache partitioning, as a *Smart Cache*. The details on the hash function are proprietary in nature.

[7] The cache partition refill time is the time to fetch from memory to the L3 cache. Thus, it is hardly affected by the fact that the Intel i7's core-to-L3 access time varies from 26 to 31 cycles. Our WCET estimation covers such L3 access variations.

Fig. 5. Page coloring on the Intel i7-2600 L3 cache

B. Taskset

Table I shows four periodic tasks that we have created for the evaluation. The task functions are selected from the PARSEC benchmark suite [7] to create a taskset consisting of cache-sensitive and cache-insensitive tasks. We utilize them as representative components of complex real-time embedded applications such as sensor fusion and computer vision in an autonomous vehicle [17]. Each task has a relative deadline D_i equal to its period T_i and a memory requirement M_i that consequently determines the minimum required number of cache/memory partitions p for the task. Task priorities are assigned by the deadline-monotonic scheduling policy.

For the WCET analysis, we used the measurement-based approach. To reduce inaccuracies in measurement, we disabled the processor's simultaneous multithreading and dynamic clock frequency scaling. All unrelated system services such as GUI and networking were also disabled during the experiments. We used the processor's hardware performance counters to measure the task execution time and the L3 misses, when each of the tasks were running alone in the system. Then, we chose the maximum observed execution time and the maximum observed L3 misses as the WCET estimate and the worst-case L3 misses, respectively. Figure 6 shows each task's per-period execution time as the number of assigned cache partitions increases. In each sub-figure, the WCET and the average-case execution time (ACET) are plotted as a solid line and a dotted line, respectively. The worst-case L3 misses per period are presented as a bar graph with the scale on the right y-axis.

The taskset used in our evaluation is a mixture of cache-sensitive and cache-insensitive tasks.[8] We can confirm this from Figure 6. τ_1 and τ_3 are cache-sensitive tasks. The τ_1's WCET C_1^p drastically decreases as the number of cache partitions p increases, until p exceeds 12. The number of τ_1's L3 misses also decreases as p increases. τ_3's WCET C_3^p continuously decreases as p increases. In terms of utilization, the difference between the maximum and the minimum utilization of τ_1 is $(C_1^{32} - C_1^1)/T_1 = 10.82\%$. The utilization difference of τ_3 is 11.83%. On the other hand, τ_2 and τ_4 are cache-insensitive. The utilization differences of τ_2 and τ_4 are merely 0.56% and 0.54%, respectively.

C. Cache Reservation

The purpose of this experiment is to verify how effective cache reservation is in avoiding inter-core cache interference.

[8]The cache sensitivity of a task is not necessarily related to whether the task is CPU-bound or memory-bound. From the cycle-per-instruction (CPI) point of view, τ_3 can be considered memory-bound because its CPI ranges from 1.23 to 1.90. The other tasks can be considered CPU-bound because their CPIs are less than 1. More details on this issue can be found in [15].

TABLE I
TASKSET INFORMATION

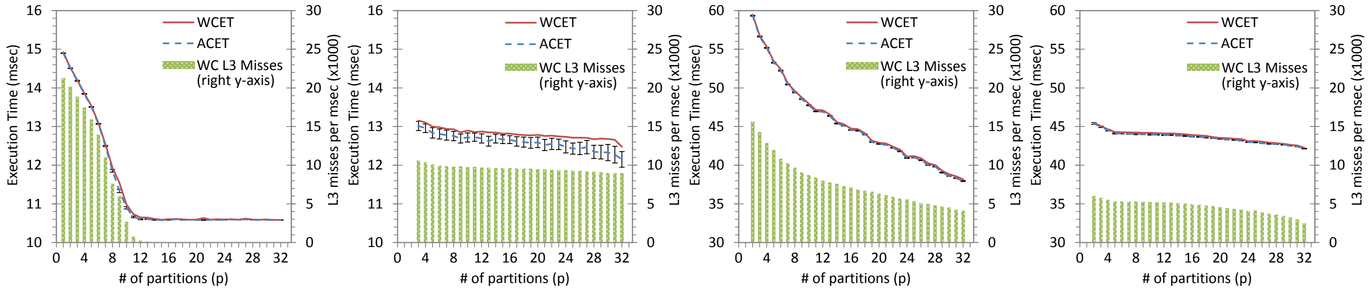
Task τ_i	T_i=D_i (msec)	M_i (MB)	Min. p	Cache Sensitive	Name and description
τ_1	40	18	1	Yes	*p_streamcluster*: computes clustering of data points
τ_2	120	66	3	No	*p_ferret*: image-based similarity search engine
τ_3	180	52	2	Yes	*p_canneal*: graph restructuring for low routing cost
τ_4	600	50	2	No	*p_fluidanimate*: simulates fluid motion for animations

We ran each task on different cores simultaneously, i.e. τ_i on Core i, under two cases: with and without cache reservation. Memory reservation was used in both cases. Without cache reservation, all tasks competitively used the entire cache space. With cache reservation, the number of cache partitions for each core was as follows: 12 partitions for Core 1, 3 for Core 2, 14 for Core 3, and 3 for Core 4. These numbers are determined to reduce the total CPU utilization by the observation of Figure 6. The cache partitions assigned to each core were solely used by the task on that core.

Figure 7 presents the observed execution time and the L3 misses of four tasks with and without cache reservation, when they ran simultaneously on different cores. In each sub-figure, the upper graph shows the execution time of each task instance and the lower graph shows the number of L3 misses for each instance. The x-axis on each graph indicates the instance numbers of a task. Tasks are released at the same instance using `hrtimers` in Linux.

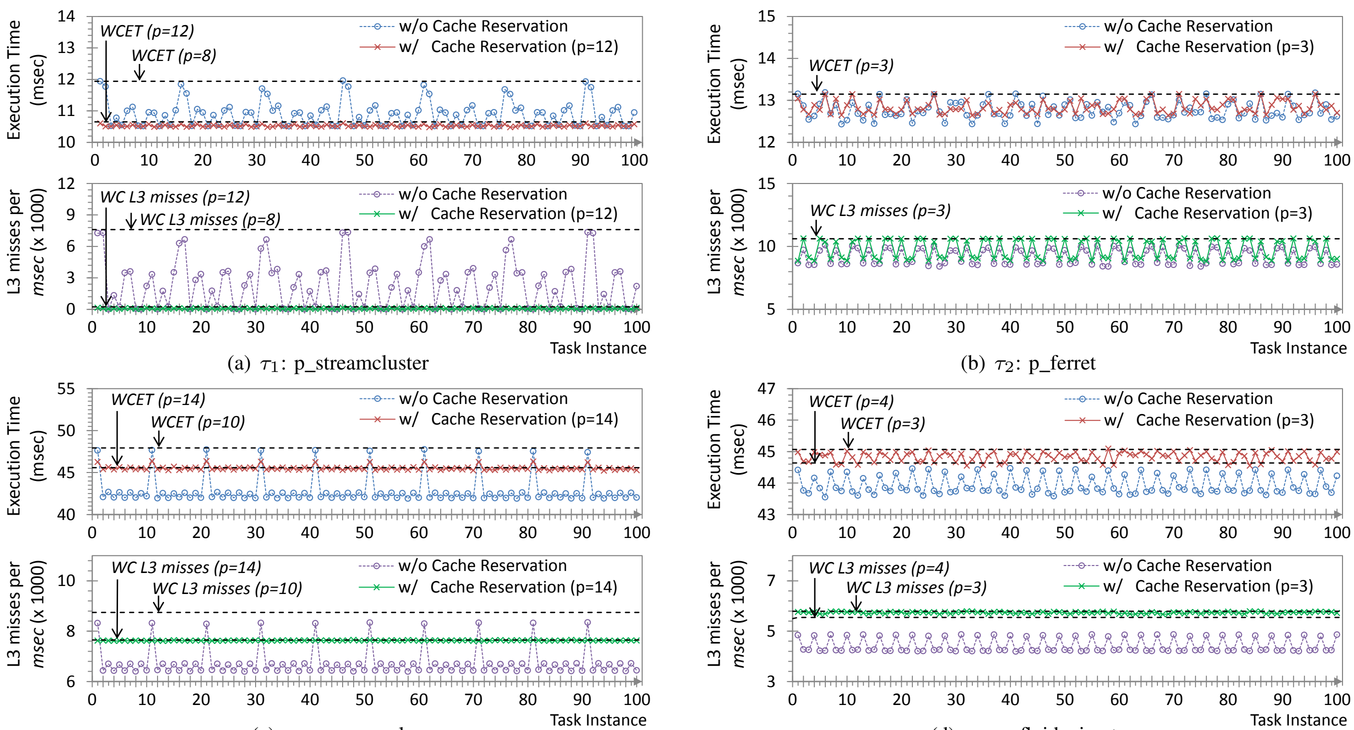
The execution times of all tasks without cache reservation vary significantly compared to the execution times with cache reservation. Without cache reservation, tasks compete for the L3 cache and higher worst-case L3 misses are encountered. The correlation between execution time and L3 misses is clearly shown in Figure 7(a) and Figure 7(c). The average execution time of tasks without cache reservation may not be much higher. However, the absence of cache reservation contributes to poor timing predictability. The longest execution time of τ_1 without cache reservation is close to its WCET with 8 dedicated cache partitions (C_1^8), that of τ_2 is close to C_2^3, that of τ_3 is close to C_3^{10}, and that of τ_4 is close to C_4^4. Note that, without cache reservation, the longest execution times cannot be obtained before profiling the whole taskset. Hence, the profiling may need to be re-conducted whenever a single parameter of the taskset changes. In addition, without cache reservation, the cache is not effectively utilized. The total number of cache partitions for the above longest execution times is $8 + 3 + 10 + 4 = 25$. This means that 7 partitions are wasted in terms of WCET.

With cache reservation, the execution times of τ_1, τ_2, and τ_4 do not exceed their WCETs that are estimated in isolation from other tasks. τ_3 also does not exceed its WCET except at the beginning of each hyper-period of 1800 msec. τ_3 exceeds its WCET by less than 2% once in a hyper-period. However, this is not caused by inter-core cache interference. As shown in Figure 7(c), the L3 misses of τ_3 instances are always lower than its worst-case L3 misses even at the beginning of each hyper-period, meaning that cache reservation successfully avoids inter-core cache interference. Since all task instances



(a) τ_1: p_streamcluster (b) τ_2: p_ferret (c) τ_3: p_canneal (d) τ_4: p_fluidanimate

Fig. 6. Execution time and L3 misses of each task as the number of cache partitions increases when running alone in the system



(a) τ_1: p_streamcluster (b) τ_2: p_ferret (c) τ_3: p_canneal (d) τ_4: p_fluidanimate

Fig. 7. Observed execution time and L3 misses of tasks when each task runs simultaneously on different cores

TABLE II
CACHE ALLOCATION TO TASKS WITH CACHE SHARING

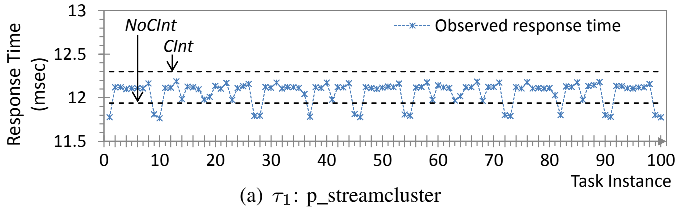
τ_i	Allocated cache partitions $S(i)$	WCET (msec)	Worst-Case Response-Time	
			NoCInt	CInt
τ_1	{1, 2, 3, 4, 5, 6, 7, 8}	C_1^8 = 11.94	11.94	12.30
τ_2	{1, 2, 3}	C_2^3 = 13.15	25.09	25.72
τ_3	{1, 2, 3, 4, 5, 6, 7, 8}	C_3^8 = 49.58	98.55	101.36
τ_4	{4, 5, 6, 7, 8}	C_4^5 = 44.30	179.88	273.78

start their execution concurrently at the beginning of each hyper-period, we strongly suspect that the observed execution time slightly greater than the WCET is caused by other shared resources on a multi-core processor, such as the memory controller and the bus memory. We plan to study this as part of our future work.

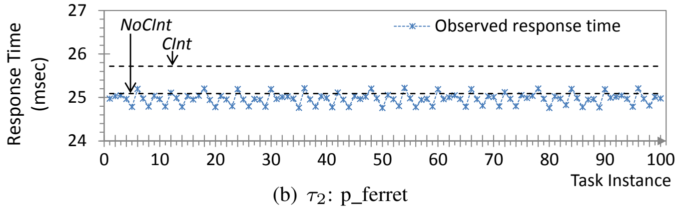
D. Cache Sharing

We first evaluate the effectiveness of our proposed equations in predicting the worst-case response time (WCRT) of a task with cache sharing. In this experiment, all tasks run on a single core with 8 cache partitions. Table II shows the cache partition allocations to the tasks by the cache-sharing technique and the predicted WCRT of the tasks. The WCRT is calculated with two methods: "*NoCInt*" means intra-core cache interference is not taken into account, and "*CInt*" means the WCRT is calculated by Equation (2).

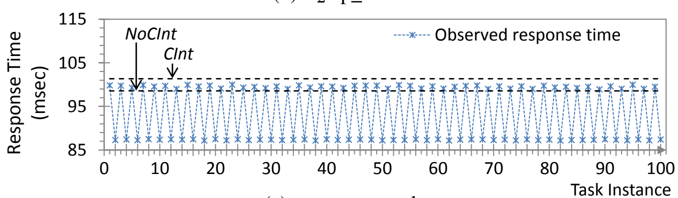
Figure 8 illustrates the observed response time of each task. The WCRT values with *NoCInt* and *CInt* are depicted as straight lines in each graph. In all tasks, the observed response time exceeds the WCRT with *NoCInt*, but does not exceed the WCRT with *CInt*. For τ_1, the observed response time greater than the WCRT with *NoCInt* is solely caused by the cache warm-up delay, because τ_1 has the highest priority task and does not experience any cache-related preemption delay. Figure 9 supports this observation. It shows the observed L3 misses of τ_1's instances. Since τ_1 shares its cache partitions with other tasks, the observed L3 misses are higher than the worst-case L3 miss value that is estimated when τ_1 does not share cache partitions. The correlation between τ_1's observed response time and observed L3 misses is also clearly shown. Hence, we can identify that τ_1's observed response time greater than the WCRT with *NoCInt* is caused by increased



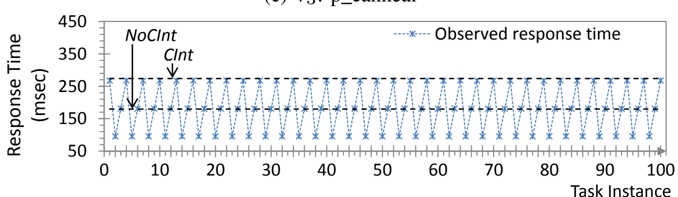
(a) τ_1: p_streamcluster



(b) τ_2: p_ferret



(c) τ_3: p_canneal



(d) τ_4: p_fluidanimate

Fig. 8. Response time of tasks with cache sharing on a single core

L3 cache misses, rather than jitter. This result shows the effect of our response time test that explicitly considers the cache warm-up delay. Task τ_4 shows a significant 93.9 msec difference between *NoCInt* and *CInt*. Since the WCRT with *NoCInt* is close to the period of τ_3, timing penalties from intra-core cache interference make the response time exceed the period of τ_3. Then, the next instance of τ_3 preempts τ_4, thereby increasing the response time of τ_4 significantly.

Secondly, we identify the utilization benefit of the cache-sharing technique by comparing the total CPU utilization with and without cache sharing. Without cache sharing, cache allocations are as follows: τ_1 is assigned 1 partition, τ_2 is assigned 3 partitions, τ_3 is assigned 2 partitions, and τ_4 is assigned 2 partitions. Note that this is the only possible cache allocation without cache sharing because the number of available cache partitions is eight, which is equal to the sum of each task's minimum cache requirement. With cache sharing, the same cache allocations as in the Table II are used. Figure 10 depicts the total CPU utilization with and without cache sharing. The left three bars are the predicted and the observed values without cache sharing and the right four bars are the values with cache sharing. The utilization values with cache sharing are almost 10% lower than the values without cache sharing. This result shows that cache sharing is very beneficial for saving the CPU utilization. Furthermore, with cache sharing, both the worst-case and the average-case observed utilization are higher than the predicted utilization with *NoCInt* but lower than the predicted value with *CInt*. This implies that Equation (1) provides a safe upper bound on the total utilization with intra-core cache interference.

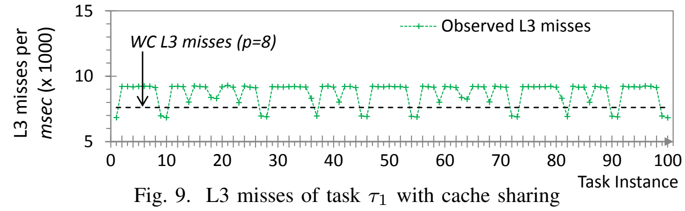


Fig. 9. L3 misses of task τ_1 with cache sharing

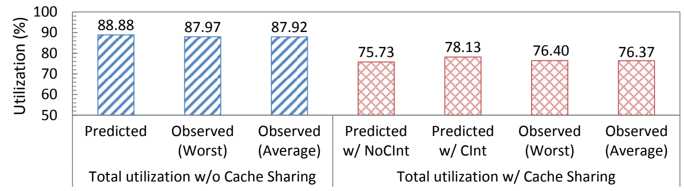


Fig. 10. Total CPU utilization with and without cache sharing

E. Cache-Aware Task Allocation

We now evaluate the effectiveness of our cache-aware task allocation (CATA) algorithm that exploits cache reservation and cache sharing. Note that it is not appropriate to compare CATA against previous approaches such as in [6][27], since (i) they do not consider the task memory requirements, which is essential to prevent page swapping when page coloring is used, and (ii) they require non-preemptive EDF scheduling due to the lack of intra-core cache interference analysis. Hence, for comparison, we consider the best-fit decreasing (BFD) and the worst-fit decreasing (WFD) bin-packing algorithms. BFD and WFD is each combined with a conventional software cache partitioning approach. Before allocating tasks, BFD and WFD evenly distribute given cache partitions to all N_C cores and sort tasks in decreasing order of task utilization with the number of per-core cache partitions. During task allocation, they do not use cache sharing.

The system parameters used in this experiment are as follows: the number of tasks $n = \{8, 12, 16\}$, the number of cores $N_C = 4$, the number of total cache partitions $N_P = 32$, and the size of total system memory $M_{total} = \{1024, 2048\}$ MB. To generate more than the four tasks in Table I, we have duplicated the taskset such that the number of tasks is a multiple of four.

We first compare in Figure 11 the minimum number of cache partitions required to schedule a given taskset under BFD, WFD, and CATA. The y-axis represents the cache partition usage as a percentage to N_P, for ease of comparison. CATA schedules given tasksets by using 16% to 25% and 12% to 19% less cache partitions than BFD and WFD, respectively. All algorithms consume more cache partitions when $M_{total} = 1024$, compared to when $M_{total} = 2048$, due to the task memory requirements. BFD fails to schedule a taskset with 16 tasks when $M_{total} = 1024$ but schedules the taskset when $M_{total} = 2048$. We next compare the memory space efficiency of the algorithms at their minimum cache partition usage. The memory space efficiency in our context is the ratio of the total memory usage of tasks to the size of allocated memory partitions, computed as $(\sum M_i)/\{(M_{total}/N_P) \times$ (# of allocated memory partitions)$\}$. Figure 12 shows the memory space efficiency. CATA is 25% to 39% and 14% to 35% more memory space efficient than BFD and WFD, respectively. Since BFD and WFD suffer from the memory co-partitioning problem, they exhibit poor memory space effi-

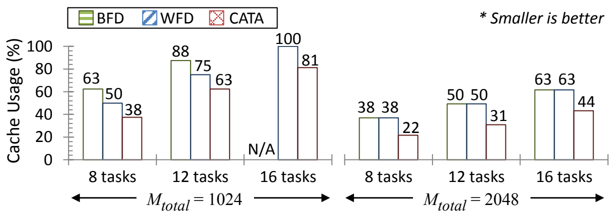


Fig. 11. Minimum amount of cache required to schedule given tasksets

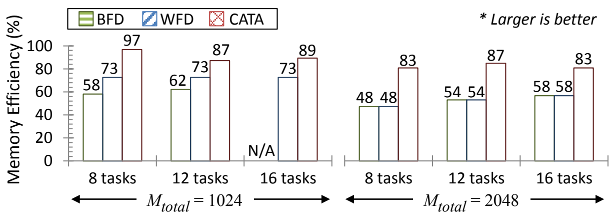


Fig. 12. Memory space efficiency at minimum cache usage

ciency. On the other hand, CATA shows 97% of memory space efficiency when $n = 8$ and $M_{total} = 1024$, meaning that only 3% of slack space exists in the allocated memory partitions. Lastly, we compare in Figure 13 the total accumulated CPU utilization required to schedule given tasksets under BFD, WFD, and CATA when all cache partitions are used. CATA requires 29% to 44% and 14% to 49% less CPU utilization than BFD and WFD, respectively. The utilization benefit of CATA becomes larger as the number of tasks increases. This is because CATA utilizes cache sharing but BFD and WFD suffer from the availability of a limited number of cache partitions. Based on these results, we therefore conclude that our scheme efficiently allocates cache partitions to tasks and significantly mitigates the memory co-partitioning problem and the availability of a limited number of cache partitions.

V. Conclusions

In this paper, we have proposed a coordinated OS-level cache management scheme for a multi-core real-time system. While providing predictable performance on architectures with shared caches across cores, our scheme addresses the two major challenges of page coloring: the memory co-partitioning problem and the availability of only limited number of cache partitions. Our scheme also yields a very noticeable utilization benefit compared to the traditional approaches. Our experimental results show the practical impact of our proposed schemes on a multi-core platform. Our scheme can be used not only for developing new multi-core real-time systems but also for migrating existing applications from single-core to multi-core platforms.

Our work focused on interference due to the presence of a shared cache, which in turn can cause significant degradation in the predictable run-time performance of a multi-core real-time system. However, there also exist other factors contributing to unexpected timing penalties in a multi-core system, such as memory bank conflicts and memory bus contention. We plan to study these issues in the future.

References

[1] AMD64 architecture programmer's manual. http://www.amd.com.

[2] Curtiss-Wright's Intel Core i7 single board computers for aerospace and defense applications. http://www.cwcdefense.com.

[3] Intel 64 and IA-32 architectures software developer's manual. http://intel.com.

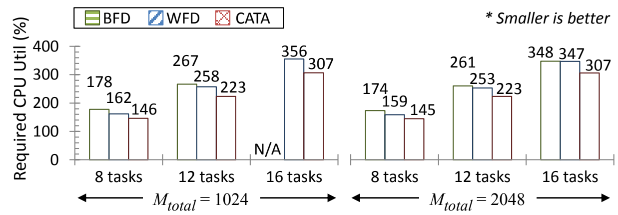


Fig. 13. Total CPU utilization required to schedule given tasksets

[4] S. Altmeyer, R. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[5] S. Basumallick and K. Nilsen. Cache issues in real-time systems. In *ACM Workshop on Language, Compiler, and Tools for Real-Time Systems*, 1994.

[6] B. Berna and I. Puaut. Pdpa: period driven task and cache partitioning algorithm for multi-core systems. In *International Conference on Real-Time and Network Systems (RTNS)*, 2012.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[8] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[9] J. Busquets-Mataix, J. Serrano, and A. Wellings. Hybrid instruction cache partitioning for preemptive real-time systems. In *Euromicro Workshop on Real-Time Systems (ECRTS)*, 1997.

[10] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *International Symposium on Microarchitecture (MICRO)*, 2006.

[11] A. Eswaran and R. Rajkumar. Energy-aware memory firewalling for QoS-sensitive applications. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

[12] X. Fu et al. Cache-aware utilization control for energy efficiency in multi-core real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2011.

[13] N. Guan et al. Cache-aware scheduling and analysis for multicores. In *ACM International Conference on Embedded Software (EMSOFT)*, 2009.

[14] M. Joseph and P. K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[15] H. Kim, A. Kandhalu, and R. Rajkumar. Coordinated cache management for predictable multi-core real-time systems. Technical report, Carnegie Mellon University, 2013. http://www.contrib.andrew.cmu.edu/~hyoseunk.

[16] H. Kim and R. Rajkumar. Shared-page management for improving the temporal isolation of memory reservations in resource kernels. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2012.

[17] J. Kim, H. Kim, K. Lakshmanan, and R. Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

[18] C.-G. Lee et al. Bounding cache-related preemption delay for real-time systems. *Software Engineering, IEEE Transactions on*, 27(9):805 –826, 2001.

[19] O. Lempel. 2nd generation intel core processor family: Intel core i7, i5 and i3. In *Hot Chips: A Symposium on High Performance Chips (HC23)*, 2011.

[20] J. Liedtke et al. OS-controlled cache predictability for real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[21] J. Lin et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[22] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[23] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis. Integrating cache related pre-emption delay analysis into edf scheduling. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2013.

[24] S. Manegold. The calibrator (v0.9e), a cache-memory and TLB calibration tool. http://www.cwi.nl/~manegold/Calibrator.

[25] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in linux. In *IEEE Real-Time Systems Symposium (RTSS) Work-In-Progress*, 1998.

[26] M. Paolieri et al. Hardware support for wcet analysis of hard real-time multicore systems. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[27] M. Paolieri, E. Quinones, F. Cazorla, R. Davis, and M. Valero. IA^3: An interference aware allocation algorithm for multicore hard real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2011.

[28] R. Pellizzoni et al. A predictable execution model for cots-based embedded systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2011.

[29] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[30] D. Tam, R. Azimi, L. Soares, and M. Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, 2007.

[31] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM J. Res. Dev.*, 46(1):5–25, 2002.

[32] A. Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2(3):315–327, 1994.

[33] M.-K. Yoon, J.-E. Kim, and L. Sha. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *IEEE Real-Time Systems Symposium (RTSS)*, 2011.

[34] X. Zhang et al. Towards practical page coloring-based multicore cache management. In *ACM European Conference on Computer Systems (EuroSys)*, 2009.