Holistic resource allocation for multicore real-time systems

Meng Xu*, Linh Thi Xuan Phan*, Hyon-Young Choi*, Yuhan Lin†, Haoran Li‡, Chenyang Lu‡, Insup Lee*
*University of Pennsylvania, †Northeastern University,‡Washington University in St. Louis

***Abstract*—This paper presents CaM, a holistic cache and memory bandwidth resource allocation strategy for multicore real-time systems. CaM is designed for partitioned scheduling, where tasks are mapped onto cores, and the shared cache and memory bandwidth resources are partitioned among cores to reduce resource interferences due to concurrent accesses. Based on our extension of LITMUS^{RT} with Intel's Cache Allocation Technology and MemGuard, we present an experimental evaluation of the relationship between the allocation of cache and memory bandwidth resources and a task's WCET. Our resource allocation strategy exploits this relationship to map tasks onto cores, and to compute the resource allocation for each core. By grouping tasks with similar characteristics (in terms of resource demands) to the same core, it enables tasks on each core to fully utilize the assigned resources. In addition, based on the tasks' execution time behaviors with respect to their assigned resources, we can determine a desirable allocation that maximizes schedulability under resource constraints. Extensive evaluations using real-world benchmarks show that CaM offers near optimal schedulability performance while being highly efficient, and that it substantially outperforms existing solutions.**

I. Introduction

Multicore processors are becoming increasingly common in real-time systems. In the automotive domain, for instance, they have been used to consolidate features that are traditionally implemented on separate electronic control units to reduce size, weight, and power consumption. This trend, however, also makes it much more difficult to guarantee timing predictability: since the cores share the last-level cache and the memory bandwidth, tasks running concurrently on different cores may interfere with one another through these resources. As a result, traditional resource allocation techniques that consider only CPU resource can no longer be safely applied.

One effective approach to mitigating interferences among concurrent tasks is *resource partitioning*: by dividing the cache and memory bandwidth among the cores, tasks running on each core can have exclusive accesses to the resources assigned to that core. This approach can be implemented using today's hardware features and memory management techniques—for instance, the shared cache can be efficiently allocated to cores using cache coloring or Intel's Cache Allocation Technology (CAT), and the memory bandwidth can be distributed among cores using regulation mechanisms such as [72]. If we combine this with a partitioned scheduling algorithm, e.g., partitioned Earliest Deadline First, timing guarantees can be achieved by applying existing schedulability analysis for uniprocessors.

However, two research questions remain in realizing this approach: (1) how to partition tasks onto the cores? and (2) how much cache and memory bandwidth should each core have? One solution is to evenly divide the cache and the memory bandwidth among the cores, and then use an existing bin-packing technique to map tasks onto cores. This method works in principle, but it has important limitations: First, not all cores need the same cache size and the same amount of memory bandwidth, since the resource demands of a core depend on that of the specific tasks running on it; therefore, evenly distributing the resources among cores could lead to inefficient use of resources. Second, existing task partitioning techniques do not consider other types of resources besides CPU, and they typically assume a fixed worst-case execution time (WCET) for each task. As our experiments in Section III show, the WCET of a task highly depends on the cache and memory bandwidth resources it is given. Without considering this behavior, the resulting mapping may result in poor timing performance, because cache- and memory-sensitive tasks may take much longer to execute if not given sufficient resources, whereas computation-intensive tasks may be given more resources than they strictly require. Prior work has considered cache and memory bandwidth in scheduling [68], but it focuses on soft real-time performance instead of schedulability.

In this paper, we take a holistic approach towards answering these research questions. Rather than decoupling them, we compute the mapping of tasks and the allocation of shared resources to cores in an integrated resource allocation strategy called CaM that considers the demands on CPU, cache, and memory bandwidth concurrently to minimize resources while ensuring timing guarantees. Developing such a strategy turns out to be highly challenging. Due to the interactions between the different types of resources, there is a tradeoff between the demand of one type and the allocation of another. For instance, when a task is allocated more cache space, it will typically incur fewer cache misses and thus have fewer memory requests, resulting in less memory bandwidth demand. Conversely, when a task is allocated more memory bandwidth, it will have a smaller average worst-case memory request latency and thus a smaller cache miss latency; as a result, it will become less sensitive to the amount of cache space it is allocated. Since all three types of resources are limited, finding the right tradeoff is non-trivial. Further, to best assign a task to a core, we need to know its CPU demand, but this demand depends on the task's WCET and thus the cache space and the amount of memory bandwidth given to its assigned core. This circular dependency makes the allocation non-trivial.

To address the above challenges, we first experimentally investigate the interdependence between the WCET of a task and the cache space and memory bandwidth it is given. For this, we integrate both the cache partitioning (using Intel's CAT) and the memory regulation (using MemGuard [72]) mechanisms into an existing real-time OS (LITMUS^{RT} [13]) to provide an end-to-end platform that can be used to experimentally explore the above relationship for real-world workloads. By exploiting this relationship, our allocation strategy can

effectively find the right tradeoff between WCET and shared resource needs. In addition, by grouping tasks with similar characteristics to a core, it enables the tasks to fully utilize the resources allocated to their assigned cores. Our evaluation shows that our strategy is highly effective in utilizing resources and maximizing schedulability—it performs very close to an optimal solution based on constraint optimization, and it substantially outperforms both a baseline solution (which distributes the resources evenly) and MC^2 (a state-of-the-art resource allocation technique developed in [17]).

In summary, we make the following contributions:

- an extensive empirical evaluation of the impact cache and memory bandwidth resources on the WCET of a task;
- CaM, an efficient and effective resource allocation algorithm for tasks that can minimize resources while guaranteeing schedulability; and
- an extensive performance evaluation of our algorithm using real-world benchmarks.

For performance evaluation, we also develop an optimal solution of our resource allocation problem using Mixed Integer Programming (MIP), which can be found in [63].

II. Related work

Memory bandwidth resource. Memory bandwidth regulation has been studied extensively at both hardware and software levels. Hardware-based techniques [23, 24, 25, 33, 75] can provide fine-grained memory bandwidth allocation to the cores using the memory controller; however, they require hardware customization and cannot be applied to COTS platforms. In contrast, software-based techniques [6, 71, 72, 73] extend the system software to implement the memory bandwidth allocation; typically, they use the performance monitoring unit to monitor the memory requests from each core, and throttle a core when it exceeds its allocated memory bandwidth. Our experimental platform leverages MemGuard [72], a state-of-the-art software-based regulation technique in real-time systems for the memory bandwidth regulation.

There exists an alternative line of work that focuses on memory bandwidth-aware timing analysis [18, 27, 50, 51, 52, 70]. These techniques account for the overhead caused by memory bandwidth interference and its impact on schedulability. Without any isolation, it is generally difficult to obtain a tight bound on the interference and thus these techniques typically produce pessimistic results. Recent research [40, 43, 48] has started to study the worst-case memory request latency in the context of memory bandwidth regulation. We expect that the results therein can be incorporated into CaM allocation strategy to consider the extra memory access delay from cores with limited memory bandwidth.

Cache resource. Several cache partitioning techniques have been proposed to reduce the shared cache interference [9, 21, 22, 26, 28, 59, 61, 62, 68, 74]. The software-based approach reorganizes a task's memory layout to allocate a specific cache area to the task using, e.g., page coloring [28, 39, 67] or compiler-based [42] techniques. The hardware-based approach [39, 61] leverages the cache partitioning capability in recent COTS processors, such as the Cache Allocation Technology (CAT) in Intel processors [5] and the Lockdown-by-Master (LbM) technology in ARM processors [2]. We use the hardware-based approach (specifically, Intel's CAT) to achieve the per-core cache allocation as it is much more efficient than the software-based approach. We note that although the interference due to concurrent accesses to the shared cache can be mitigated using cache partitioning, tasks running on the same core may still experience cache-related preemption and migration delay (CRPMD) overhead. Our work assumes that such overhead is included in the WCET of a task, but it can easily be extended to account for such overhead in the allocation, e.g., by incorporating with existing CRPMD overhead-aware analysis [8, 14, 19, 61].

Recent research [60] has also considered the shared cache interference directly in the analysis instead of cache partitioning. While promising, the initial result is limited to direct-mapped noninclusive cache (which is rarely used in COTS multicore processors) and thus requires further research to make it more practically applicable.

Multiple resources. There also exists a large body of work on management schemes that consider multiple types of resources concurrently [12, 35]. These techniques consider different combinations of resources, such as the co-allocation of cache and memory bank resources [35], or the co-allocation of cache and memory bandwidth resources [12, 32, 53, 54, 58]. The majority of these techniques focus on improving the average performance (e.g., throughput) and fairness of the system, and thus cannot be directly applied to our setting.

The management of multiple resources has recently been studied for latency-sensitive systems. For instance, Heracles [37] can manage CPU, cache, memory bandwidth, and networking resources altogether to achieve low latency for latency-sensitive tasks in data centers, such as web search and online machine learning clustering algorithm. However, Heracles considers a highly simplified real-time setting where each machine has *only one* latency-sensitive task. In contrast, our work solves the resource co-allocation for a much more general real-time task setting. Ma et al [38] proposed PARD, a new programmable hardware architecture to improve QoS and resource utilization. However, PARD requires customized hardware support and is not suitable for COTS platforms.

Recent research (e.g., [55, 66]) in the real-time community has begun to investigate the impact of multiple resources on real-time performance. Researchers in [6, 41, 47, 66] have developed coordinated allocation schemes for CPU and memory bandwidth resources, but they ignored the cache resource. Researchers in [30, 31, 55] considered the cache and memory bank resources but ignored the memory bandwidth resources. Researchers in [17] proposed a multi-resource allocation algorithm for real-time mixed criticality systems, which focuses on the tradeoffs of the cache allocation while considering the memory bank allocation. The allocation algorithm in [17] has two assumptions: (i) tasks have been pre-allocated to cores (using a bin-packing algorithm); and (ii) memory banks are

evenly allocated to cores.[1] Thus, it focuses on the allocation of only one resource (i.e., cache resource) and does not consider the allocation tradeoffs between multiple resources. Unlike these techniques, our work considers all three types of resources (CPU, cache, and memory bandwidth) in the allocation. Recently, Ye et al. [68] proposed MARACAS, a multicore scheduling and load-balancing framework to address cache and memory bandwidth interference; however, MARACAS focuses on soft real-time requirements.

Existing work has also investigated the impact of other types of shared resources on real-time performance, including e.g., impacts of Miss Status Holding Registers (MSHR) on partitioned caches [56], interferences due to memory bank contention [31, 55, 69], TLB interferences [45], and effects of interrupts [13, 29, 46, 49]. As an initial step, our work focuses on interferences due to concurrent cache accesses and shared memory bandwidth, while assuming that the overhead due to other types of interferences has been accounted for in the tasks' WCETs. We defer the extension to other types of interferences—which is highly non-trivial—to future work.

Partitioning algorithms. The problem of partitioning real-time tasks onto identical multiprocessor platforms has been known to be NP-hard. Several polynomial-time algorithms [15, 20, 44] have been proposed for solving it approximately (see [10] for a comprehensive comparison). These techniques consider only CPU resource and assume that each task has a fixed WCET; therefore, they cannot be directly applied to our setting. There exists extensions to multiple WCETs per task – for instance, energy-aware partitioning algorithms that consider a vector of WCETs for a task depending on the speed of the processor have been proposed in [7, 65]. However, algorithms in this setting cannot be apply as they typically ignore the shared cache and memory BW resources, which work differently from that of energy.

III. Impact of cache and memory bandwidth resources on WCET

We begin by presenting the experimental evaluation that guides our resource allocation strategy. Specifically, our evaluation aims to (1) investigate the benefits of cache and memory isolation on timing performance, and (2) explore the relationship between the allocated cache and memory resources and the timing behavior of a task. For this, we needed a real-time OS with built-in cache partitioning and memory bandwidth regulation features. As we were not aware of any such free open-source platform available, we extended LITMUS^{RT} with Intel's CAT and MemGuard for our experiments. Before discussing the experimental results, we first describe this integrated prototype. For convenience, we refer to memory bandwidth simply as bandwidth (BW) in the rest of the paper.

A. Prototype

The prototype integrates the Intel's CAT and MemGuard bandwidth regulation mechanisms as Linux modules into LITMUS^{RT}, a real-time extension of the Linux kernel that provides several real-time scheduling policies [16]. We used LITMUS^{RT} as the base real-time OS, since it is open-source, established, and actively maintained.

Intel's CAT. The Intel's CAT is a new hardware feature that allows system software (e.g., the OS) to control the allocation of the shared last-level cache to physical cores. It divides the shared cache into N non-overlapped equal-size cache partitions (e.g., $N = 20$ on our evaluation machine). It provides two types of model-specific registers: (1) the Class of Service (COS) register, which has an N-bit Capacity Bitmask (CBM) field to specify a particular cache partition set; and (2) the IA32_PQR_ASSOC (PQR) register, which has a COS field for linking a COS register to a core.

To allocate a set of specific cache partitions to a core, we need to modify (1) the CBM field of a COS register to specify the cache partition set and (2) the core's PQR register to link the COS register to the core. Both modifications can be performed using the *wrmsr* instruction, which can be executed from the user space using the Intel MSR Tools [1] (or Intel RDT software package [4]). To enable the operator to configure cache partitions for cores, we wrote a script that takes as inputs two parameters – the core index and the bitmask of the cache partition set for the core – and invokes the Intel MSR Tools to set the cache partitions for the core.

MemGuard. MemGuard [73] provides different BW management mechanisms, including reservation, reclaiming and best-effort bandwidth sharing. We leveraged the reservation mechanism for our prototype, as it is the only one that can achieve guaranteed per-core bandwidth reservation. We used the existing MemGuard implementation and loaded it as a Linux module in LITMUS^{RT}. We controlled BW allocation by writing to the *proc* filesystem exposed by MemGuard.

B. Experimental setup

Hardware. Our prototype ran on a machine with a CAT-capable Intel Xeon E5-2618L v3 processor with a 20MB 20-way set-associative L3 shared cache and a single channel 8GB PC-2133 DDR4 DRAM. The cache can be divided into 20 equal partitions, and a core must be allocated at least 2 partitions (due to hardware constraints). The maximum guaranteed bandwidth was $1.4GB/s$ (obtained using the same method as in [73]). For our experiments, we divided the bandwidth into 20 partitions of 70MB/s each, and the maximum bandwidth budget allocated to a core was always equal to (the size of) one or multiple partitions.

System configuration. We enabled 4 cores in BIOS as in [62] because the processor has only 4 Class of Service (COS) registers[2], supporting at most 4 cores with different cache partition settings. Like in most existing real-time research [28], we disabled hyper-threading, SpeedStep, and hardware cache prefetcher features to avoid non-deterministic timing behavior. We also shut down all non-essential system services during our experiments to minimize potential interferences.

Workload. We considered two types of workloads: (i) a benchmark workload, taken from the PARSEC benchmark

[1] We confirmed these two assumptions with the authors of [17].

[2] The number of COS registers varies across Intel processors.

suite [11] and a *cache-bench* synthetic benchmark, and (ii) an interference workload. Both the PARSEC benchmark suite and the *cache-bench* benchmark have been used as an evaluation workload in prior real-time research [27, 28, 57, 62]. Characteristics of the former (e.g., the working set size) are available in [11], while the latter is a cache-intensive program that uses a linked list to sequentially access every 64 bytes (i.e., the cache-line size) of a 20MB array. The interference workload consists of *cache-bomb* (similar to the one used in [62]), a program that uses the array index to sequentially access every 64 bytes of a 40MB array until it is terminated.

C. Evaluation of resource isolation

To evaluate how well CaM can protect a task's WCET from being affected by concurrent running tasks, we ran a (PARSEC or *cache-bench*) benchmark task on one core, and ran a *cache-bomb* task on each of the remaining cores. We measured the execution time of the benchmark task when it is allocated a fixed number of cache partitions and BW partitions. For comparision, we performed the same experiment for four more settings, with (i) cache allocation only, (ii) BW allocation only, (iii) neither cache nor BW management, and (iv) both cache and BW allocation, but without running any interference task.

The results show that cache allocation or BW allocation in isolation is often not sufficient to prevent tasks from both cache and BW interferences. In contrast, by integrating both techniques, CaM can effectively isolate tasks from both types of interferences. A detailed discussion of the experiments and results can be found in our technical report [63].

D. Impact of cache and bandwidth allocation on WCET

Experiment. For this evaluation, we ran the *canneal* benchmark on one core, which is assigned different numbers of cache partitions and BW partitions. We measured the benchmark task's execution time across 25 runs, and calculated its *resource slowdown* factor under a cache and BW configuration (as the ratio of the task's measured WCET to the WCET when it is allocated all cache and BW partitions in the system).

Results. Figs. 1 and 2 show the impact of BW and cache resource allocation on the task's WCET, respectively. Fig. 1 shows that the *canneal* benchmark task's slowdown varies from 17.06× to 2.84× when the task is allocated 1 memory bandwidth partitions; in contrast, the slowdown does not change substantially when the task is allocated 20 memory bandwidth partitions. A similar trend can also be observed in Fig. 2. In general, we can make the following observation:

Observation 1. *The relation between a task's WCET and the amount of cache (resp. memory bandwidth) resource it receives is highly dependent on the amount of BW (resp. cache) it receives. In particular, a task's WCET is more sensitive to the cache allocation when it is allocated a smaller amount of BW, and vice versa.*

This behavior is expected, as the more cache space a task receives, the fewer cache misses it incurs, and thus the frequency that it is throttled also decreases. Similarly, when a task receives less memory bandwidth, it runs out of memory budget more quickly and becomes throttled more frequently, thus making it more sensitive to its allocated cache space.

We repeated the experiment with each PARSEC benchmark to examine the effect of the workload's characteristics. Our results show that the above observed pattern varies across benchmarks. Due to space constraints, we omit the results.

Observation 2. *The relation between a task's WCET and its allocated cache and BW resources varies across different benchmark tasks. Some tasks (e.g.,* canneal *benchmark) are sensitive to both cache and BW resources, whereas others are sensitive to only one (e.g.,* facesim *benchmark) or none (e.g.,* swaptions *benchmark) of the resources.*

These results motivate the need for considering the relationship between CPU, cache and BW resources to achieve better utilization and schedulability. We formally define this co-resource allocation problem next.

IV. Theoretical modeling and problem statement

Using our prototype, we can already partition cache and BW resources among cores to provide better isolation among concurrent tasks. To meet timing guarantees, however, we also need a resource allocation strategy that, given a set of tasks, decides 1) how to map tasks onto cores, and 2) how to distribute cache and BW resources among cores, to minimize resource usage while ensuring schedulability. To solve this problem, we first formalize a concrete platform model based on our experimental platform, and a cache- and memory-aware task model based on the tasks' timing characteristics observed in our empirical evaluation. (A table that summarizes the notations is available in our technical report [63].)

Platform model. The platform consists of M identical cores, with a shared cache and a shared memory bus that are accessible by all cores. The cache is divided into N_{cp} equal-size cache partitions, and the memory bandwidth is divided into N_{bw} equal-size BW partitions. Cache and BW allocation is done at the core level: each core is allocated a distinct set of cache partitions and a certain number of BW partitions, all of which will be available to any task currently running on the core. As some hardware does not allow an allocation that is fewer than a certain number of partitions, we denote by N_{cp}^{min} and N_{bw}^{min} the minimum numbers of cache partitions and BW partitions that a core must be allocated, respectively. We assume that the OS scheduler schedule tasks using the partitioned Earliest Deadline First (EDF) policy (supported by LITMUS^{RT}), due to its high resource utilization bound.

Task model. We consider independent periodic tasks with implicit deadlines[3], but we extend it to capture the relationship between the task's WCET and the cache and BW allocation. Specifically, a task τ_i is modeled as $\tau_i = \{p_i, d_i, e_i(cp_i, bw_i) \mid N_{cp}^{min} \leq cp_i \leq N_{cp} \wedge N_{bw}^{min} \leq bw_i \leq N_{bw}\}$, where p_i is the period, $d_i (= p_i)$ is the deadline, and $e_i(cp_i, bw_i)$ is the task's WCET when it is assigned cp_i cache partitions and bw_i bandwidth partitions. Specifically, the WCET $e_i(cp_i, bw_i)$ of a task is the task's execution time when it executes *alone* on a core,

[3]We assume this for simplicity; it should be straightforward to extend the algorithm to constrained deadline tasks.

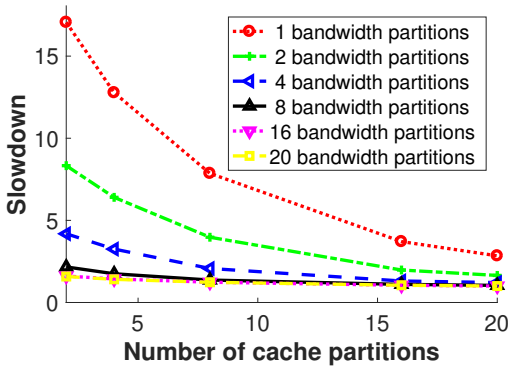


Fig. 1: Memory bandwidth impact.

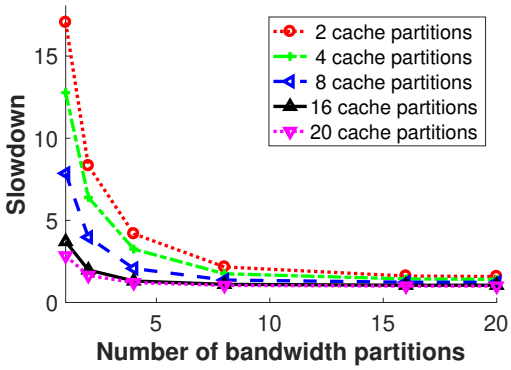


Fig. 2: Cache impact.

where the core is allocated cp_i cache partitions and bw_i bandwidth partitions. We assume that all task parameters are given a priori. (Here, the vector of WCETs $e_i(cp_i, bw_i)$ of a task τ_i can be obtained by analysis or measurement. In our evaluation, it is obtained through profiling, as was done in our experimental evaluation.) Note that, by definition, a task's WCET is invariant with respect to the tasks that are scheduled on the same core. However, a task's *response time* depends on tasks running on the same core, because of scheduling delay and various overhead such as CRPD and BW interference caused by higher-priority tasks. As a first step towards timing predictability, like in prior work [28, 61], we assume that a task's WCET values have been inflated to account for other sources of overhead; however, it can be extended to explicitly accounted for such overhead by incorporating with existing work such as [64] [48] [13].

For analysis purpose, we refer to $re_i = e_i(N_{cp}, N_{bw})$ as the *reference WCET* of τ_i, i.e., the WCET when it is allocated all cache and memory partitions in the system. We denote by $ru_i = re_i/p_i$ the *reference utilization* of τ_i, which is the utilization based on its reference WCET. By abuse of notation, we use the term *assigned WCET* and *assigned utilization* to denote the WCET and utilization of a task, respectively, when it is already assigned a fixed number of cache partitions and a fixed number of memory bandwidth partitions. We require that any task must be assigned to a core.

As usual, we say that a task is schedulable *iff* it always finishes execution before its deadline, and the system is schedulable if all tasks are schedulable. According to the EDF schedulability test [34], the tasks on a core are schedulable *iff* their total utilization does not exceed 1. The system is schedulable *iff* the tasks on each core are schedulable.

Problem statement. Given the above model, our goal is to develop a strategy for computing (i) a mapping of tasks to cores, and (ii) the number of cache partitions and the number of BW partitions (per regulation period) for each core in the system, so that the system is schedulable and the number of cores needed is minimized.

Challenge. In principle, our resource allocation problem can be solved using constraint optimization techniques. For reference, we have developed an MIP formulation of the proposed problem (details can be found in [63]). Although being optimal, this approach has a very high running time complexity and is too inefficient to be practical. Achieving both effectiveness and efficiency turns out to be highly challenging in our setting, because of the inter-dependence between WCET, cache allocation, and memory bandwidth allocation, as well as their variances across tasks. The problem we consider is, in fact, more general than the traditional problem of packing tasks to cores, which is known to be NP-hard. In the next section, we present a novel approach to solve this problem, using a combination of clustering and bin-packing heuristics.

V. Heuristic Resource Allocation Algorithm

We first discuss the basic strategies that guide our allocation and present a high-level overview of the algorithm. We then present the details of the algorithm and discuss its complexity.

A. Overview

Based on the insights obtained from the empirical evaluation in Section III-D, we propose the following high-level strategies. These strategies aim to exploit the relationship between the allocated resources and a task's WCET, as well as the diverse resource demands across tasks.

Strategy 1. *(Group by sensitivity) As tasks on the same core are always allocated the same amount of cache and BW resources (equal to that of the core), grouping tasks with similar sensitivity to cache and BW resource allocations onto the same core can help better utilize cache and BW resources.*

Our allocation strategy aims to group tasks with similar resource sensitivity onto the same core. To capture how sensitive a task is to different allocations of cache and BW resources, we define the *resource-allocation slowdown* of a task τ_j under a resource configuration (cp_k, bw_k) to be $t\_slowdown_j(cp_k, bw_k) = \frac{e_j(cp_k, bw_k)}{re_j}$, where re_j is the task's reference WCET (i.e., the WCET when the task is given all partitions; c.f. Section IV) and $e_j(cp_k, bw_k)$ is the WCET of τ_j under cp_k cache partitions and bw_k BW partitions. Each task τ_j has an N_{config} dimensional slowdown vector $\vec{sv}_j$, where N_{config} is the number of all valid resource configurations (cp_k, bw_k), and the k^{th} element of the vector is the resource allocation slowdown of τ_j under the k-th resource configuration. As the slowdown vector $\vec{sv}_j$ captures how sensitive a task is to different cache and BW configurations, we also refer to it as the *resource sensitivity* of τ_j.

The next strategy simply aims to balance load across cores; the number of cores is a parameter to our algorithm:

Strategy 2. *(Load balancing) Given an allocation of resources to tasks, evenly distributing the tasks among cores based on the assigned tasks' utilizations can help balance the load across cores and avoid under-utilized cores.*

We define the *resource utility* of a core i as the average reduced utilization per additional cache and BW partition allocated to the core:

$$\mathsf{resourceU}_i(cp,bw) = \begin{cases} (u_i - u_i')/(cp+bw) & \text{if } u_i > 1 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where u_i and u_i' are the core's utilization before and after it is given extra cp cache partitions and bw memory bandwidth partitions, respectively. In our algorithm, each core is assigned a set of tasks, and it is initially allocated the minimum amount of cache and BW resources. If a core is schedulable, its resource utility is always 0. Among the cores that are not schedulable (if any), we will allocate additional resources to the core with the largest resource utility, so that we can better use the limited resources to make all cores schedulable:

Strategy 3. *When adding more cache and BW resources to a core that is unschedulable under the current allocation, allocating resources to a core that results in the maximum* resource utility *can provide a more effective use of the limited cache and BW resources.*

Overview of the algorithm. Algorithm 1 shows the high-level idea of our allocation algorithm for m cores. Initially, each core is given the minimum number of cache and BW partitions. The algorithm then works in three phases:

(1) Phase 1 (Lines 1–2): It first groups tasks that have similar resource sensitivity (i.e., similar slowdown vector) into the same cluster, based on Strategy 1. Then, it sorts tasks in each cluster in decreasing order of tasks' reference utilization – this is because it typically is harder for a task with higher utilization to find a feasible core.

(2) Phase 2 (Lines 4–9): It randomly picks one permutation of the clusters as the order of packing clusters to cores. It then packs each task in each cluster onto cores, such that the total *reference utilization* of tasks on each core is similar (i.e., close to the average reference utilization of all cores), as guided by Strategy 2. Next, using the *allocResource()* procedure, it allocates cache and BW resources to cores to maximize the resulting resource utility, based on Strategy 3. Once the resources allocated to each core are determined, it calculates the resulting utilization of each core and checks the system's schedulability. If the resulting utilization of each core is no larger than 1, the system is schedulable. If the system is schedulable, the algorithm terminates and outputs the resource allocation policy that schedules the system; otherwise, it continues to the next phase.

(3) Phase 3 (Lines 10–18): The algorithm tries to balance the workload across cores (Line 13). For each unschedulable core, it migrates each of its tasks to a schedulable core that will have the smallest utilization after the migration, until the unschedulable core becomes schedulable. After the *balance* procedure finishes, the algorithm re-runs the *allocResource()* procedure for cores and checks if the system will become schedulable. The algorithm keeps balancing tasks on cores until the system becomes schedulable or there is no benefit in balancing (Line 16). Because the order of the clusters may affect the bin packing result (Line 5), which may later affect the resource allocation and balance procedure, the algorithm re-orders the clusters and repeats Phases 2 and 3 (Lines 3–20) for a user-specified constant number (i.e., *maxIterPerm*) before it claims that the system is unschedulable.

As discussed in the previous section, a system that is unschedulable when using m cores may become schedulable with fewer cores. This is because with fewer cores, there are more cache and BW resources available to each core on

Algorithm 1 Heuristic resource allocator

Input: V: the set of tasks, m: the number of cores, N_{cp}: the number of cache partitions, N_{bw}: the number of bandwidth partitions, *maxIterKM*: the maximum iterations for KMeans, *maxIterPerm*: the maximum iterations.
Output: Schedulable or Unschedulable.

```
1:  clusters ← clusterTasks(V, m, maxIterKM)
2:  Sort tasks in each cluster in decreasing order of their reference utilization
3:  repeat
4:      perm_clusters ← permute(clusters)  ▷ randomly pick one permutation of clusters
5:      cores ← binPackClusters(perm_clusters, m)
6:      cores ← allocResource(cores, m, N_cp, N_bw)  ▷ The cores variable specifies tasks and resources allocated to each core
7:      sched ← checkSchedulability(cores)  ▷ schedulable if each core's assigned utilization is no larger than 1
8:      if sched = schedulable then
9:          break
10:     oldVal ← ∞  ▷ previous imbalance value
11:     while true do ▷ balance cores' utilizations iteratively
12:         val ← getImbalanceValue(cores)
13:         cores ← balance(cores)
14:         cores ← allocResource(cores, m, N_cp, N_bw)
15:         sched ← checkSchedulability(cores)
16:         if sched = schedulable or val > oldVal then
17:             break
18:         oldVal ← val
19:     maxIterPerm ← maxIterPerm − 1
20: until maxIterPerm = 0
21: return sched
22:
23: function getImbalanceValue(cores)
24:     imbalance = 0
25:     for all c ∈ cores do
26:         if c's assigned utilization > 1 then
27:             imbalance += c's assigned utilization − 1
28:     return imbalance rounded to 2 fractional digits
```

average (since each active core must have at least some cache and BW partitions). This can lead to smaller tasks' utilizations (if the tasks are sensitive to cache and bandwidth resources), hence making the system easier to schedule. To obtain a feasible allocation with the minimum number of cores, we use Algorithm 1 to find a feasible allocation on every valid number of cores m, where $1 \leq m \leq M$, and M is the maximum number of cores supported by the hardware. We then use the smallest value of m for which an allocation exists and its corresponding task mapping and allocation configuration for the system.

B. Details of the algorithm

We now discuss the key ideas of the four main procedures used in our algorithm: *clusterTasks(), binPackClusters(), allocResource(), and balance()*. Due to space constraints, we present their pseudo-code in [63].

The *clusterTasks()* procedure uses the KMeans algorithm [36]—a widely used machine learning method for clustering data points with similar features—to cluster tasks that have similar sensitivity to cache and memory bandwidth resources. Recall from Section V-A that the resource sensitivity of each task τ_j is defined as its slowdown vector $\vec{sv}_j$, whose elements are the slowdown values of τ_j under different valid cache and BW resource configurations. Formally, the procedure aims to divide the set of tasks τ into m clusters, such that the pairwise deviation of tasks in the same cluster is minimized:

$$\arg\min_{C} \sum_{k=1}^{m} \frac{1}{2|C_k|} \sum_{\tau_i,\tau_j \in C_k} ||\vec{sv}_i - \vec{sv}_j||^2 \tag{2}$$

where $|C_k|$ is the number of tasks in the cluster C_k and C is the set of clusters.

The *clusterTasks()* procedure has three steps: (1) initialization, which calculates each task's slowdown vector and creates an initial set of m clusters; (2) assignment, which assigns each task to its closest cluster whose mean has the least square distance to the task; and (3) update, which calculates the new mean of each cluster as the new centroid of the cluster. The algorithm repeats the assignment step and the update step until all clusters' assigned tasks are no longer changed or until *maxIterKM* iterations have reached.

Observe that, since the *clusterTasks()* procedure does not consider load balancing during its clustering, the reference utilizations of the resulting task clusters may vary substantially. Therefore, if we map each cluster directly to a core, the cores corresponding to clusters with high reference utilizations will likely be overloaded and become unschedulable, causing the entire system to become unschedulable. To avoid this issue, we employ the *binPackCluster()* procedure to balance the workload assigned to each core.

The *binPackCluster()* procedure packs tasks of clusters into m cores, such that each core's reference utilization (i.e., the total reference utilizations of all tasks on the core) is similar. The procedure first computes the average reference utilization $\mathsf{meanRefU}$ of m clusters (i.e., the total reference utilization of all tasks divided by m). Then it uses our modified first-fit bin-packing algorithm to pack tasks to cores: for each task, it attempts to pack the task into cores, going from core 0 to core $m-1$. It packs a task to a core if the core's current reference utilization is smaller than the average reference utilization $\mathsf{meanRefU}$ and the core's current reference utilization plus the task's reference utilization is no larger than 1. The procedure packs a task to core 0 if it cannot find any core that satisfies the above condition.

The *allocResource()* procedure allocates cache and BW resources to cores, such that the system is schedulable while minimizing cache and BW resources. We define the *optimal resource utility* of a core c_i when the system has N_{cp}^{idle} unused cache partitions and N_{bw}^{idle} unused bandwidth partitions as the maximum resource utility of core c_i when it is given cp extra cache partitions and bw extra bandwidth partitions, for all $cp \leq N_{cp}^{idle}$ and $bw \leq N_{bw}^{idle}$. That is,

$$\mathsf{optResourceU}_i(N_{cp}^{idle}, N_{bw}^{idle}) = \max_{N_{cp}^{min} \leq cp \leq N_{cp}^{idle},\ N_{bw}^{min} \leq bw \leq N_{bw}^{idle}} \mathsf{resourceU}_i(cp, bw) \tag{3}$$

The *allocResource()* procedure has three steps: The initialization step allocates the minimum number of cache and BW partitions to each core. Next, the resource allocation step allocates cp cache partitions and bw bandwidth partitions to the core c that has the maximum *optimal resource utility* with N_{cp}^{idle} cache partitions and N_{bw}^{idle} bandwidth partitions available. In other words, it follows Eq. (4) to allocate some or all remaining partitions to the core that has the maximum *optimal resource utility*:

$$\arg\max_{cp,bw,c\ c_i \in C} \mathsf{optResourceU}_i(N_{cp}^{idle}, N_{bw}^{idle}) \tag{4}$$

where C is the set of cores, and cp and bw are the number of cache partitions and BW partitions, respectively, to be allocated to the core c with the currently remaining N_{cp}^{idle} idle cache partitions and N_{bw}^{idle} idle BW partitions. Finally, the update step updates the remaining amount of cache and BW resources and the utilization of the core c. The entire procedure repeats until all cores become schedulable or there is no benefit in reducing the utilization of unschedulable cores.

The *balance()* procedure migrates tasks from unschedulable cores to schedulable cores to balance the utilizations across cores, to make it easier to schedule the system. It takes as input m cores whose tasks and number of cache and memory bandwidth partitions have been determined. The procedure first sorts tasks on unschedulable cores in increasing order of tasks' assigned slowdowns (recall that the assigned slowdown of a task is its assigned utilization divided by its reference utilization). For each unschedulable core, the procedure migrates each sorted task to the core that has the smallest assigned utilization after the migration, until the unschedulable core becomes schedulable. The procedure terminates after all unschedulable cores in the input become schedulable. Note that the schedulable cores in the input may become unschedulable after the migration, in which case the heuristic algorithm will call *allocResource()* to re-allocate resources to cores (c.f. Algorithm 1).

Complexity. We first discuss the complexity of the subprocedures. The *clusterTasks()* procedure enumerates all clusters for all tasks for *maxIterKM* iterations; hence, it takes

$\mathcal{O}(N \cdot m \cdot maxIterKM)$ time. The *binPackCluster()* procedure iterates over all tasks in each cluster (i.e., N tasks in total) for each core, which takes $\mathcal{O}(N \cdot m)$ time. The *allocResource()* procedure takes $\mathcal{O}(N_{cp} \cdot N_{bw})$ to calculate Eq 3 for each core and iterates for maximum $N_{cp} \cdot N_{bw}$ times, thus it takes $\mathcal{O}(m \cdot {N_{cp}}^2 \cdot {N_{bw}}^2)$ time. The *balance()* procedure sorts at most N tasks and iterates at most N tasks for at most m cores, so it takes $\mathcal{O}(N \cdot \log N) + \mathcal{O}(N \cdot m)$ time.

We next discuss the complexity of the entire heuristic algorithm by analyzing Algorithm 1. In the loop of the balancing operation (Line 13 to Line 16), the algorithm invokes the *balance()* and *allocResource()* procedures for at most 100 times, because *imbalance*'s value decreases by at least 0.01 for each loop and the algorithm stops the loop if *imbalance*'s value does not decrease. In each iteration (Line 3 to Line 20), the heuristic algorithm invokes the *binPackCluster()* and *allocResource()* procedures once, and it invokes the loop of the balancing operation once. The heuristic algorithm invokes the iteration for at most *maxIterPerm* times (Line 20). The algorithm's complexity is determined by the longest path in the algorithm: $\mathcal{O}(N \cdot m \cdot maxIterKM) + \mathcal{O}(maxIterPerm \cdot \max\{N \cdot m, N \cdot \log N, m \cdot {N_{cp}}^2 \cdot {N_{bw}}^2\})$.

VI. Numerical performance evaluation

To evaluate the effectiveness and efficiency of our heuristic resource allocation algorithm, we conducted an extensive set of experiments using randomly generated real-time workloads. We had three main objectives: (i) to evaluate the performance of our algorithm in terms of schedulability; (ii) to investigate the impact of platform configurations and task parameters on the schedulability performance; and (iii) to evaluate the efficiency of our algorithm. For comparison, we performed the same set of experiments for three other solutions: an MIP-based optimal algorithm [63]; a baseline algorithm that combines existing bin-packing heuristics with evenly distributing cache and BW resources to cores; and a variant of MC^2, a state-of-the-art resource allocation algorithm proposed in [17].

A. Experimental setup

Workload. Each workload contained a number of randomly generated periodic tasksets. The tasks' reference utilizations followed one of the three uniform distributions, whose utilizations were distributed uniformly over $[0.01, 0.1]$ (light), $[0.1, 0.4]$ (medium), and $[0.4, 0.9]$ (heavy). The tasks' workloads were selected randomly from the PARSEC benchmarks [11]. A task's WCET values under different cache and BW configurations were assigned to be the same as the WCET values of the corresponding benchmark, which were obtained by profiling using our experimental prototype on a real machine. A task's period was assigned to be the ratio of its reference WCET to its reference utilization.

We profiled the WCETs of different PARSEC benchmarks with the *simlarge* input type under different resource configurations using CaM prototype on the machine we used for our empirical evaluation (c.f. Section III). For each PARSEC benchmark with each type of input, we dedicated one core for the benchmark, configured the core with a valid cache and BW configuration, and measured the WCET of the benchmark for 25 runs. The valid number of cache partitions was ranged from 2 to 20, with a step of 1; the valid number of BW partitions was ranged from 1 to 20, with a step of 1. The set of valid resource configurations is the cartesian product of the valid number of cache partitions and the valid number of BW partitions. For each PARSEC benchmark with each type of input, we measured its WCETs under $19 \times 20 = 380$ valid resource configurations. The obtained WCETs were used for the tasks, as explained above.

Platform configurations. We analyzed the above generated workloads for two platform configurations, which are based on the Intel Xeon 2618v3 (Platform A) and Intel Xeon D-1518 (Platform B) processors we have available: Platform A has 4 cores and 20 cache partitions; Platform B has 4 cores and 12 cache partitions. The number of BW partitions is the same as the number of cache partitions on each platform.

Baseline algorithm. The baseline algorithm evenly distributes cache and BW resources to cores. Each core has N_{cp}/M cache partitions and N_{bw}/M bandwidth partitions, where N_{cp} is the total number of cache partitions, N_{bw} is the total number of BW partitions, and M is the number of cores on the platform. The WCET of a task is the measured WCET of the corresponding benchmark under N_{cp}/M cache partitions and N_{bw}/M bandwidth partitions. The algorithm uses bin-packing algorithms to pack tasks into cores. If the total assigned utilization of tasks on each core is no larger than 1, the system is deemed schedulable; otherwise, it is deemed unschedulable. We considered three bin-packing approaches: first-fit, best-fit, and worst-fit. For each task set, we ran all three approaches and selected the best one as the result (i.e., if the task set was schedulable by at least one of the approaches, we considered it as schedulable by the baseline algorithm).

MC^2 algorithm. As we are not aware of any prior work that solves the resource allocation of cache and BW resources concurrently, we selected the MC^2 algorithm [17] for comparison because it is the closest state-of-the-art algorithm to ours. As MC^2 considers cache and *memory bank* allocation, it cannot be directly applied and thus we modified it for our setting as follows: (1) we pre-allocated tasks to cores by using the same bin-packing algorithm used in [17]; (2) we evenly allocated the BW partitions to cores, just as the original MC^2 algorithm does for memory bank resources; and (3) we used the same MILP in [17] to determine the cache allocations for tasks. As mixed-criticality is out of scope of this work, we considered only level-B criticality in MC^2 and used partitioned EDF for task scheduling. We refer to this modified algorithm as MC^2.

Variances of our heuristic algorithm. The heuristic algorithm we propose consists of two phases: clustering and resource allocation. To evaluate the impact of each phase on the overall performance of the algorithm, we considered two variances of our algorithm: (1) Heuristic/CL algorithm, which skips the clustering phase; and (2) Heuristic/RA algorithm, which skips the resource allocation phase.

Analysis. We analyzed the same set of tasksets with each of

the algorithms considered above. For our heuristic algorithm, we considered three settings of $(maxIterKM, maxIterPerm)$, including $(100,24)$, $(50,12)$, and $(200,48)$. As the performance results are consistent across these settings, we present only the results for $(100,24)$ due to space constraints.

Our analyses were performed on an Intel Xeon E5-2620 v3 processor, which has 24 cores (with hyper threading enabled) operating at 2.40GHz. We set the analysis timeout value as 8 hours[4]. We refer to a taskset for which an analysis timed out as an incomputable taskset for the analysis.

B. Schedulability performance

We generated tasksets with taskset utilization ranging from 1 to 4, with a step of 0.1. For each taskset utilization, we generated 50 independent tasksets (i.e., 1550 tasksets in total), with tasks' utilizations uniformly distributed in $[0.1, 0.4]$. We analyzed the tasksets for Platform A using all the algorithms.

Relative performance of the different algorithms. Fig. 3 shows the fraction of schedulable tasksets out of 1550 tasksets under each algorithm. The number of schedulable tasksets under Optimal, Heuristic, MC^2, and Baseline algorithms are 1247, 1146, 576 and 551, respectively. The results show that the fraction of schedulable tasksets under our heuristic algorithm was very close to that of the optimal algorithm: only 8.10% of the tasksets (i.e., $1247 - 1146 = 101$ out of 1247) were deemed schedulable by the optimal solution but deemed unschedulable under our algorithm. Further, our algorithm significantly outperformed both baseline and MC^2 algorithms: it was able to schedule about twice many tasksets ($1146/551 = 2.08\times$ and $1146/576 = 1.99\times$) compared to the baseline and MC^2 algorithms, respectively. We also observe that all tasksets that were schedulable under the baseline and MC^2 algorithms were also schedulable under our heuristic algorithm in our experiments.

Impact of the clustering and resource allocation phases. The impact of the clustering and resource allocation phases on our heuristic algorithm can be observed based on the relative schedulability performance of the heuristic algorithm and its two variances, Heuristic/CL and Heuristic/RA, respectively (see also Fig. 3). The results show that, while Heuristic could schedule 1146 tasksets, Heuristic/CL and Heuristic/RA could only schedule 1037 and 463 tasksets, respectively. This suggests that (1) both phases are critical to the performance of our heuristic algorithm, and that (2) the resource allocation phase has a more positive impact on the overall performance of our heuristic algorithm than the clustering phase does, since omitting the resource allocation phase (i.e., Heuristic/RA) leads to substantially fewer schedulable tasksets.

C. Impact of platform configurations and task parameters

Impact of platform configurations. We investigated the impact of the platform configurations on the fraction of schedulable tasksets for all four algorithms. For this, we repeated the above experiment on another platform (i.e., Platform B) and reported the results in Fig. 5.

[4]We thought 8 hours (i.e., a typical work day) is sufficiently large to deem an algorithm that does not complete within the timeout impractical.

We observe that, again, our heuristic algorithm consistently performed very close to the optimal solution on both platforms. On Platform B, the optimal algorithm was able to schedule 673 tasks, while our heuristic algorithm could schedule 638 tasksets (i.e., only $1 - 638/673 = 5.20\%$ reduction). Our heuristic algorithm also substantially outperformed the baseline and MC^2 algorithms: it could schedule $3.39\times$ ($= 638/188$) and $2.24\times$ ($= 638/285$) more tasksets than the baseline algorithm and the MC^2 algorithm did, respectively.

Impact of task parameters. We investigated the impact of the task parameters on the fraction of schedulable tasksets for all four algorithms. We repeated the experiment in Section VI-B using tasksets with uniform-heavy and uniform-light utilization distributions. The results are shown in Fig. 4.

We observe that our heuristic algorithm continued to perform close optimal across different distributions of tasks' utilizations. For the tasksets with uniform-heavy utilization distribution, the optimal algorithm had 1081 schedulable tasksets, while our heuristic algorithm had 1057 schedulable tasksets, which is only $1 - 1057/1081 = 2.22\%$ fewer.

TABLE 1: Number of analyzed tasksets (out of 1550 tasksets) for uniform-light tasksets on Platform A.

	Heuristic	Optimal	Baseline	MC^2
Schedulable	1154	713	537	513
Unschedulable	396	56	1013	1037
Incomputable	0	781	0	0

We further observe that for the tasksets with uniform-light utilization distribution, the optimal algorithm started to time out after 8-hour computation for a taskset when the taskset utilization was larger than 1.9, while our heuristic algorithm never timed out. Table 1 summarizes the number of schedulable tasksets, unschedulable tasksets and incomputable tasksets for all four algorithms for the uniform-light tasksets on the Platform A. Note that, even in the *unlikely* situation where all incomputable tasksets had turned out to be schedulable for the optimal algorithm, our heuristic algorithm still performed close to optimal: at most 21.94% of the tasksets, i.e., (713 + 781 - 1154) out of 1550, would be deemed schedulable by the optimal solution but unschedulable under our algorithm. It can also be observed that our algorithm outperformed the baseline algorithm and the MC^2 algorithm by a significant factor: it was able to schedule more than twice as many tasksets compared to the baseline and MC^2 algorithms (1154/537 = $2.15\times$ and 1154/513 = $2.25\times$, respectively). These results demonstrate that not only is our heuristic algorithm substantially more efficient than the optimal solution but it is also highly effective in allocating resources.

D. Running time efficiency

We measured the running time of four algorithms in the evaluation in Fig. 3, under which the optimal algorithm never times out.[5] The result is shown in Fig. 7. We observed that our algorithm is highly efficient: its maximum average running time was 0.90 minute. On the contrary, the optimal algorithm's running time increased exponentially as the taskset's utiliza-

[5]This running-time measurement favors the optimal algorithm.

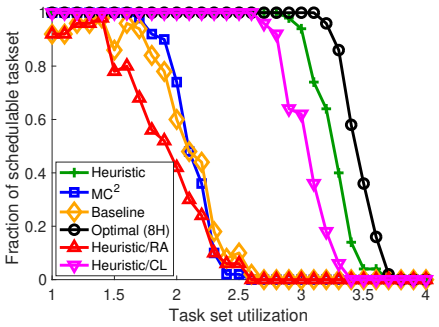
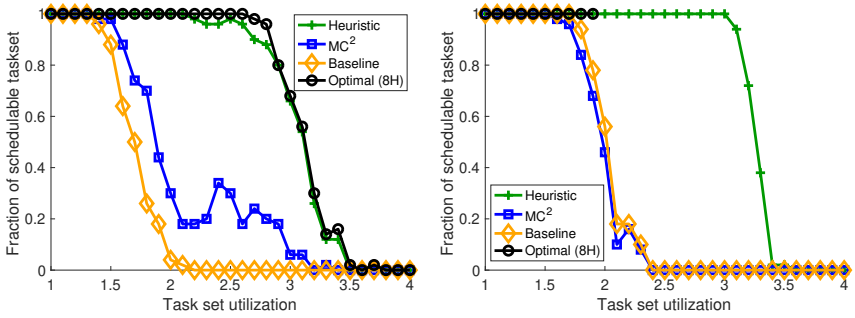


Fig. 3: Performance on Platform A.



(a) Uniform-heavy. (b) Uniform-light.

Fig. 4: Impact of task utilization distributions.

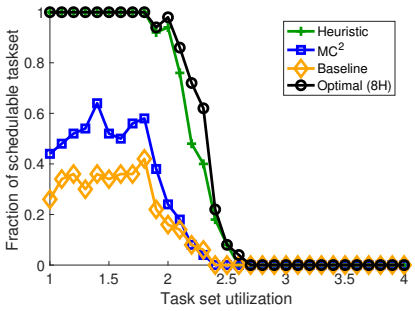


Fig. 5: Performance on Platform B.

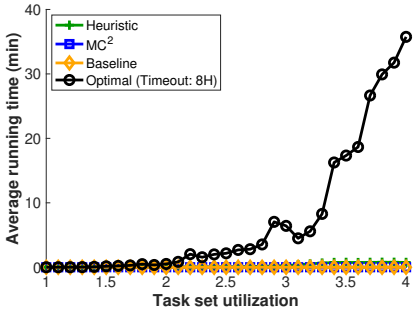


Fig. 6: Average computation time.

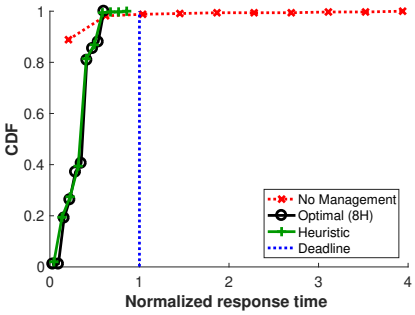


Fig. 7: Normalized response time of jobs.

tion increased, and its maximum average running time was 35.75 minutes, i.e., $35.75/0.90 = 39.72\times$ slower than ours.

VII. Experimental Performance Evaluation

To demonstrate the utility of CaM and to validate its performance experimentally, we ran a set of PARSEC benchmarks with real-time parameters in our experimental platform (described in Section III). We used the task mapping and allocation configuration computed by our heuristic allocation algorithm (Heuristic). For comparison, we also ran the same taskset with the task mapping and allocation configurations computed by two other settings: the optimal algorithm (Optimal) on our prototype; and and a baseline (vanilla) that uses first-fit bin-packing algorithm to pack tasks onto cores in LITMUS^{RT} (without cache or memory bandwidth management support).

Workload. We first converted the PARSEC benchmarks into LITMUS^{RT}-compatible real-time tasks. We then randomly generated real-time tasks whose utilizations are uniformly distributed in [0.1, 0.4] until we obtained a taskset with reference utilization of 2.0. We also used one *cache-bomb* (which was used in Section III) as a background task.

Experiment. Under the heuristic and optimal settings, we reserved one core, two cache partitions and one BW partition for the background task, and we computed the system configuration for the real-time tasks with the rest of resources. Under the vanilla setting, we reserved one core for the background task but we did not control cache or bandwidth resources. We ran the taskset for 2 minutes for each setting and used the feather-trace [3] to collect the response time of real-time jobs.

Results. Fig. 7 shows the Cumulative Distribution Function (CDF) plot of the normalized response time of all real-time tasks' jobs under the three different settings. The normalized response time of a job is the ratio of its observed response time to its relative deadline. In Fig. 7, the vertical blue line (marked as Deadline) shows the time when the normalized response time is 1. The data points that fall to the right of this line correspond to the jobs that missed their deadlines.

We observe that all jobs met their deadlines under the heuristic setting. In addition, the response times of jobs when using the heuristic algorithm were near to the values obtained under the optimal setting. In contrast, not all jobs met their deadlines under the vanilla setting, and some jobs experienced unreasonably long response time (multiple times the deadline). The results validate that CaM can effectively manage cache and memory shared resources to reduce interference and improve the timing performance on real multicore platforms.

VIII. Conclusion

We have presented a resource allocation strategy for real-time multicore systems that considers CPU resource demand of a task and the allocation of cache and memory bandwidth resources in a holistic manner. Our strategy integrates existing cache partitioning and memory bandwidth regulation mechanisms to enable the co-allocation of both resources. Through insights from our empirical evaluation of real workloads on real hardware, we designed an effective and efficient algorithm that exploits the interdependence relationship between the cache and BW resources and the tasks' WCETs in its allocation. We have shown through extensive evaluations that our strategy can effectively reduce interference, and that it offers near optimal schedulability performance while being highly efficient.

ACKNOWLEDGEMENT

This research was supported in part by ONR N00014-16-1-2195, NSF CNS 1703936, CNS 1563873 and CNS 1750158, and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-16-C-0056 and HR0011-17-C-0047.

REFERENCES

[1] MSR-tools. https://01.org/msr-tools. Accessed: 2016-02-01.

[2] PrimeCell level 2 cache controller (PL310) - technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0246c/index.html. Accessed: 2015-03-29.

[3] Tracing with LITMUS^{RT}. http://www.cs.unc.edu/~anderson/litmus-rt/doc/tracing.html. Accessed: 2015-10-15.

[4] User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat. Accessed: 2017-01-10.

[5] x86: Intel Cache Allocation Technology support. http://lwn.net/Articles/622893/. Accessed: 2015-01-09.

[6] A. Agrawal, G. Fohler, J. Freitag, J. Nowotsch, S. Uhrig, and M. Paulitsch. Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study. In *ECRTS*, 2017.

[7] B. N. Alahmad and S. Gopalakrishnan. Energy efficient task partitioning and real-time scheduling on heterogeneous multiprocessor platforms with qos requirements. *Sustainable Computing: Informatics and Systems*, 1(4):314 – 328, 2011.

[8] S. Altmeyer, R. I. Davis, and C. Maiza. Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *RTSS*, 2011.

[9] M. A. Awan, K. Bletsas, P. F. Souto, B. Akesson, and E. Tovar. Mixed-Criticality Scheduling with Dynamic Redistribution of Shared Cache. In *ECRTS 2017*, 2017.

[10] S. Baruah. Partitioned edf scheduling: A closer look. *Real-Time Syst.*, 49(6):715–729, Nov. 2013.

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[12] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *MICRO*, 2008.

[13] Björn B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[14] R. J. Bril, S. Altmeyer, M. M. Heuvel, R. I. Davis, and M. Behnam. Fixed priority scheduling with pre-emption thresholds and cache-related pre-emption delays: Integrated analysis and evaluation. *Real-Time Syst.*, 53(4):403–466, July 2017.

[15] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, Dec 1995.

[16] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *RTSS*, 2006.

[17] M. Chisholm, W. Bryan C, N. Kim, and J. H. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS*, 2015.

[18] D. Dasari, V. Nelis, and B. Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Syst.*, 52(3):272–322, May 2016.

[19] R. I. Davis, S. Altmeyer, and J. Reineke. Analysis of write-back caches under fixed-priority preemptive and non-preemptive scheduling. In *RTNS*, pages 309–318, New York, NY, USA, 2016. ACM.

[20] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Oper. Res.*, 26(1):127–140, Feb. 1978.

[21] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven llc allocation. In *USENIX ATC*, 2016.

[22] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT*, 2009.

[23] D. Guo and R. Pellizzoni. A requests bundling dram controller for mixed-criticality systems. In *RTAS*, 2017.

[24] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *RTAS*, 2015.

[25] M. Hassan, H. Patel, and R. Pellizzoni. PMC: A requirement-aware dram controller for multicore mixed criticality systems. *ACM Trans. Embed. Comput. Syst.*, 16(4):100:1–100:28, May 2017.

[26] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li. A simple cache partitioning approach in a virtualized environment. In *ISPA*, 2009.

[27] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *RTAS*, 2014.

[28] H. Kim and R. R. Rajkumar. Real-time cache management for multi-core virtualization. In *EMSOFT*, 2016.

[29] H. Kim, S. Wang, and R. Rajkumar. Responsive and enforced interrupt handling for real-time system virtualization. In *RTCSA*, Aug 2015.

[30] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *RTAS*, 2017.

[31] N. Kim, B. C. Ward, M. Chisholm, C.-Y. F. , J. H. A. , and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS*, 2016.

[32] B. Li, L. Zhao, R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell. Coqos: Coordinating qos-aware shared resources in noc-based socs. *Journal of Parallel and Distributed Computing*, 71(5):700 – 713, 2011.

[33] Y. Li, B. Akesson, and K. Goossens. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Syst.*, 52(5):675–729, Sept. 2016.

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

[35] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu. Going vertical in memory management: Handling multiplicity by multi-policy. In *ISCA*, 2014.

[36] S. P. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[37] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[38] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *ASPLOS*, 2015.

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

[40] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In *ECRTS*, 2017.

[41] A. Melani, M. Bertogna, V. Bonifaci, A. Marchetti-Spaccamela, and G. Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *RTNS*, New York, NY, USA, 2015. ACM.

[42] F. Mueller. Compiler support for software-based cache partitioning. In *LCTES*, 1995.

[43] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.

[44] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Syst.*, 9(3):207–239, nov 1995.

[45] S. A. Panchamukhi and F. Mueller. Providing task isolation via tlb coloring. In *RTAS*, April 2015.

[46] P. Patel, M. Vanga, and B. B. Brandenburg. Timershield: Protecting high-priority tasks from low-priority timer interference (outstanding paper). In *RTAS*, April 2017.

[47] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for cots-based embedded systems. In *RTAS*, Washington, DC, USA, 2011. IEEE Computer Society.

[48] R. Pellizzoni and H. Yun. Memory servers for multicore systems. In *RTAS*, 2016.

[49] L. T. X. Phan, M. Xu, J. Lee, I. Lee, and O. Sokolsky. Overhead-aware compositional analysis of real-time systems. In *RTAS*, April 2013.

[50] S. Schliecker and R. Ernst. Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embed. Comput. Syst.*, 10(2):22:1–22:27, Jan. 2011.

[51] A. Schranzhofer, J.-J. Chen, and L. Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *RTAS*, 2010.

[52] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo.

Timing analysis for resource access interference on adaptive resource arbiters. In *RTAS*, 2011.

[53] A. Sharifi, S. Srikantaiah, A. K. Mishra, M. Kandemir, and C. R. Das. Mete: Meeting end-to-end qos in multicores through system-wide resource management. *SIGMETRICS Perform. Eval. Rev.*, 39(1):13–24, June 2011.

[54] L. Subramanian. *Providing High and Controllable Performance in Multicore Systems Through Shared Resource Management*. PhD thesis, Carnegie Mellon University, 2015.

[55] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *ICESS*, Washington, DC, USA, 2013.

[56] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, April 2016.

[57] P. K. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS*, 2016.

[58] X. Wang and J. F. Martinez. Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *HPCA*, 2015.

[59] X. Wang, X. Wen, Y. Li, Z. Wang, Y. Luo, and X. Li. Dynamic cache partitioning based on hot page migration. *Frontiers of Computer Science*, 6(4):363–372, 2012.

[60] J. Xiao, S. Altmeyer, and A. D. Pimentel. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *RTSS*, 2017.

[61] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS*, 2016.

[62] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. vcat: Dynamic cache management using cat virtualization. In *RTAS*, 2017.

[63] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *Technical Report*, 2019. http://www.cis.upenn.edu/~linhphan/papers/rtas19-CaM-techreport.pdf.

[64] M. Xu, L. T. X. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *RTSS*, 2013.

[65] C. Yang, J. Chen, T. Kuo, and L. Thiele. An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems. In *DATE*, April 2009.

[66] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, Sept 2016.

[67] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *PACT*, 2014.

[68] Y. Ye, R. West, J. Zhang, and Z. Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *RTSS*, 2016.

[69] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, 2014.

[70] H. Yun, R. Pellizzoni, and P. K. Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *ECRTS*, 2015.

[71] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, 2012.

[72] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, 2013.

[73] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, Feb 2016.

[74] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *EuroSys*, 2009.

[75] Y. Zhou and D. Wentzlaff. MITTS: Memory inter-arrival time traffic shaping. In *ISCA*, 2016.