Holistic resource allocation for multicore real-time systems

Meng Xu  
University of Pennsylvania, mengxu@seas.upenn.edu

Linh T.X. Phan  
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Part of the Computer and Systems Architecture Commons, Computer Sciences Commons, and the Systems and Communications Commons

Recommended Citation

This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/1018
For more information, please contact repository@pobox.upenn.edu.
Holistic resource allocation for multicore real-time systems

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 LITMUSRT 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 the resource constraints. Extensive evaluation using real-world benchmarks show that CaM offers near optimal schedulability performance while being highly efficient, and that it substantially outperforms baseline solutions that combine bin-packing with an equal distribution of resources among cores.

Keywords
real-time multicore, cache and DRAM management, resource allocation

Disciplines
Computer and Systems Architecture | Computer Engineering | Computer Sciences | Systems and Communications

This technical report is available at ScholarlyCommons: https://repository.upenn.edu/cis_reports/1018
Holistic resource allocation for multicore
real-time systems

Meng Xu\textsuperscript{1}, Linh Thi Xuan Phan\textsuperscript{2}, Hyon-Young Choi\textsuperscript{3}, and Yuhan Lin\textsuperscript{4}

\textsuperscript{1} University of Pennsylvania, Philadelphia, U.S.
\texttt{mengxu@cis.upenn.edu}
\textsuperscript{2} University of Pennsylvania, Philadelphia, U.S.
\texttt{linhphan@cis.upenn.edu}
\textsuperscript{3} University of Pennsylvania, Philadelphia, U.S.
\texttt{hyonchoi@cis.upenn.edu}
\textsuperscript{4} Northeastern University, Shenyang, P.R. China
\texttt{linyuhan@stumail.neu.edu.cn}

\textbf{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\textsuperscript{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 the resource constraints. Extensive evaluation using real-world benchmarks show that CaM offers near optimal schedulability performance while being highly efficient, and that it substantially outperforms baseline solutions that combine bin-packing with an equal distribution of resources among cores.

\textbf{1998 ACM Subject Classification} C.3 Real-Time and Embedded Systems

\textbf{Keywords and phrases} real-time multicore, cache and DRAM management, resource allocation

\textbf{Digital Object Identifier} 10.4230/LIPIcs.ECRTS.2018.YY

\textbf{Acknowledgements.} We would like to thank Insup Lee for the insightful discussion on the resource allocation algorithm.

\textbf{1 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, which can help to reduce cost, 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 shared resources. As a result, traditional scheduling and analysis techniques that consider only CPU resource can no longer be safely applied.
One effective approach to mitigating or removing the interferences among concurrent tasks is resource partitioning: by dividing the cache and memory bandwidth among the different 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 [57]. If we combine this with a partitioned scheduling algorithm, such as partitioned Earliest Deadline First (pEDF), 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 amount of memory bandwidth, since the resource demands of a core depend on that of the specific tasks running on it. Thus, 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 3 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.

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 algorithm that considers the CPU demands and the cache and memory bandwidth resources concurrently. Prior work has considered cache and memory bandwidth in scheduling [54], but it focuses on soft real-time performance instead of schedulability. Developing an effective task and resource allocation strategy that minimizes resources for hard timing guarantees turns out to be highly challenging. First, 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; as a result, the task will require less memory bandwidth. Likewise, 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 also become less sensitive to amount of cache space it is allocated. Since all three types of resources are limited, finding the right tradeoff is non-trivial. Second, the assignment of a task to a core requires us to know the task’s 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 a lot harder.

In this paper, we present a novel resource allocation strategy called CaM that considers cache and memory resources and CPU demands in a holistic manner. 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 [57]) mechanisms into an existing real-time OS (LITMUSRT [11]) 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, to minimize resources. In addition, by grouping tasks with similar characteristics to a core, it enables the tasks to fully utilize the resources allocated to their assigned core. Our evaluation shows that our strategy is highly effective in utilizing
resources and maximizing schedulability—its performance is very close to that of an optimal solution and substantially better than that of a baseline solution (which distributes the resources evenly).

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;
- an optimal solution for the resource allocation based on Mixed Integer Programming (MIP);
- an efficient and effective resource allocation algorithm for tasks that can minimize resources while guaranteeing schedulability; and
- an extensive evaluation of our algorithm.

**Organization of the paper.** We begin with a review of related work, followed by the empirical evaluation that explores the impact of cache and memory bandwidth allocation on WCET in Section 3. Based on the observed timing behavior, we introduce a concrete model for the resource allocation problem and present an MIP-based solution in Sections 4 and 5. We discuss our resource allocation strategy in Section 6, and report performance evaluation results in Section 7.

## 2 Related work

### Memory bandwidth resource.
Memory bandwidth regulation has been studied extensively at both hardware and software levels. Hardware-based techniques [18–20, 28, 60] can provide fine-grained memory bandwidth allocation to the cores by monitoring and scheduling memory requests from each core using the memory controller. These techniques require hardware customization and cannot be applied to COTS platforms. In contrast, software-based techniques [7, 56–58] extend the system software to implement the 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 bandwidth. Our experimental platform uses MemGuard [57], a state-of-the-art software-based regulation technique in real-time systems for the bandwidth regulation.

There exists an alternative line of work that focuses on memory bandwidth-aware timing analysis [14, 22, 40–42, 55]. 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; thus, these techniques typically produce pessimistic results. Recent research [37, 39] 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 [16, 17, 21, 23, 47, 49, 50, 54, 59]. The software-based approach reorganizes a task’s memory layout to allocate a specific cache area to the task using, e.g., page coloring [23, 34, 53] or compiler-based [36] techniques. The hardware-based approach [34, 49] 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.

Recent research [48] 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 non-inclusive cache (which is rarely used in COTS multicore processors) and thus requires further research to make it more practically applicable.

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
Holistic resource allocation for multicore real-time systems

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, 12, 15, 49].

Multiple resources. There also exists a large body of work on management schemes that consider multiple types of resources concurrently [10, 30]. These techniques consider different combinations of resources, such as the co-allocation of cache and memory bank resources [30], or the co-allocation of cache and memory bandwidth resources [10, 27, 43, 44, 46]. The majority of these techniques focus on improving the average performance (e.g., throughput) and fairness of the system, and thus cannot be applied to hard real-time systems.

The management of multiple resources has recently been studied for latency-sensitive systems. For instance, Heracles [32] 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 [33] 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., [45, 52]) in the real-time community has begun to investigate the impact of multiple resources on real-time performance. Researchers in [7, 35, 38, 52] have developed coordinated allocation schemes for CPU and memory bandwidth resources, but they ignored the cache resource. Researchers in [24, 26, 45] considered the cache and memory bank resources but ignored the memory bandwidth resources. Unlike these techniques, our work considers all three types of resources (CPU, cache, and memory bandwidth) in the allocation. Recently, Ye et al. [54] proposed MARACAS, a multicore scheduling and load-balancing framework to address cache and memory bandwidth interference; however, MARACAS focuses on soft real-time requirements.

3 Impact of cache and memory bandwidth resources on WCET

In this section, we present our 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 regulation features. As we were not aware of any such free open-source platform available, we extended LITMUS\textsuperscript{RT} with Intel’s CAT and MemGuard for our experiments. Before discussing our experimental results, we first describe this integrated prototype.

3.1 Prototype

The prototype integrates the Intel’s CAT and the MemGuard memory bandwidth regulation mechanisms as Linux modules into LITMUS\textsuperscript{RT}, a real-time extension of the Linux kernel that provides several real-time scheduling policies [13]. We used LITMUS\textsuperscript{RT} as the base real-time OS as 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, the system software performs two steps: (1) it modifies the CBM field of a COS register to specify the specific cache partition set; and (2) it
modifies the PQR register of the core to link the COS register to the core. To modify these two types of model-specific registers, we can use the \textit{wrmsr} instruction, which must execute in privilege level.

To execute the \textit{wrmsr} instruction from the user space, we need the Intel MSR Tools [1], which provides system operators with utilities to access the processor MSRs. To enable the operators to configure cache partitions for cores, we wrote a script, which takes two parameters from system operators: the core index and the bitmask of the cache partition set for the core. The script invokes the Intel MSR Tools to set the cache partitions for the core. \footnote{An alternative approach to control cache partitions for cores is to use Intel RDT software package [4].}

\textbf{MemGuard.} MemGuard [58] provides three memory bandwidth management mechanisms: (i) reservation, which guarantees that a core gets at most its allocated memory bandwidth; (ii) reclaiming, which uses a memory usage predictor to predict a core’s memory bandwidth usage in the next period, and donates a core’s predicted unused memory bandwidth to other cores; (iii) best-effort bandwidth sharing management, which further utilizes the additional achieved bandwidth above the guaranteed bandwidth. Since our work focuses on hard real-time systems, we used the first mechanism to achieve guaranteed per-core bandwidth reservation. We used the MemGuard implementation and loaded it as a Linux module in LITMUS\textsuperscript{RT}. We controlled memory bandwidth allocation by writing to the \textit{proc} filesystem exposed by MemGuard.

\section{Experimental setup}

\textbf{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 [58]. 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.

\textbf{System configuration.} We enabled 4 cores in BIOS as in [50] because the processor has only 4 Class of Service (COS) registers, supporting at most 4 cores with different cache partition settings. \footnote{The number of COS registers varies across Intel processors.} Like in most existing real-time research [23], we disabled hyper-threading, SpeedStep, and hardware cache prefetcher features to avoid non-deterministic timing behavior. To minimize interference with the experimental workload, we shut down all non-essential system services during our experiments.

We ran the tasks under evaluation on one core and interference tasks on the other three cores.

\textbf{Workload.} We considered two types of workloads: the PARSEC benchmark suite [9] and an interference workload. For the PARSEC benchmarks, we used \textit{simsmall} as the default input. For the interference workload, we have \textit{cache-bomb} (similar to the one used in [50]), which uses array index to sequentially access every 64 bytes (i.e., the cache-line size) of a 40MB array until it is terminated.

\section{Evaluation of resource isolation}

\textbf{Experiment.} To evaluate how well CaM can protect a task’s WCET from being affected by concurrent running tasks, we ran a PARSEC benchmark task on core, and ran a \textit{cache-bomb} task on each of the other three cores. We allocated 18 cache partitions and 17 bandwidth partitions to the benchmark core (i.e., which executed the benchmark task), shared the remaining 2 cache partitions for the other three cores and allocated 1 memory bandwidth partition to each of the other three cores. We measured the WCET of the benchmark task across 50 runs, which we refer to as \textit{PolluteCAM}. For comparison,
we conducted the same experiment for four additional settings: (i) the **PolluteCA** setting, where we only managed the cache; (ii) the **PolluteBW** setting, where we only managed the memory bandwidth; (iii) the **Pollute** setting, where all tasks shared the entire cache and memory bandwidth without any management; and (iv) the **Alone** setting, where we managed the cache and memory bandwidth and did not run any interference task.

![Figure 1](image)

**Figure 1** Measured WCETs of PARSEC benchmarks.

**Results.** Fig. 1 shows the slowdown factor of each PARSEC benchmark task for the five settings. The slowdown factor in a setting is the ratio of the benchmark task’s WCET to its WCET obtained in the **Alone** setting. We have the following observations from Fig. 1.

First, the execution time of the benchmark task in the **Pollute** setting is substantially larger than that of the **Alone** setting (up to $1.38 \times$ in **c anneal**). This demonstrates that cache and memory bandwidth interference can impede a task’s schedulability because of the increased WCET. Second, the execution times of some benchmark tasks (e.g., **facesim** and **c anneal**) in the **PolluteCA** setting and in the **PolluteBW** setting are still substantially larger than that of the **Alone** setting (up to $1.08 \times$). This demonstrates that cache partitioning or memory bandwidth partitioning in isolation is not sufficient to prevent such tasks from both cache and memory bandwidth interferences. Finally, the execution time of the benchmark task in the **PolluteCAM** setting is close to that in the **Alone** setting (at most 1% larger in **facesim**). This demonstrates that CaM can effectively isolate tasks from both cache and memory bandwidth interferences, and that the benefit of cache partitioning and memory bandwidth partitioning can be accumulated. This observation also suggests that we can profile tasks’ WCETs in isolation and compose them when integrating in the same system.

### 3.4 Impact of cache and memory bandwidth allocation on WCET

**Experiment.** To evaluate the impact of cache and memory bandwidth allocation on WCET, we ran the **c anneal** benchmark on one core, and we configured the core with different numbers of cache partitions and memory bandwidth partitions. We measured the benchmark task’s execution time across 25 runs, and calculated its **resource slowdown** factor under a cache and bandwidth allocation configuration (as the ratio of the measured task’s WCET to its WCET when it is allocated all available cache and memory bandwidth partitions in the system).

**Results.** Figs. 2 and 3 show the impact of memory bandwidth and cache resource allocation on the task’s WCET, respectively. Fig. 2 shows that the **c anneal** benchmark task’s slowdown varies from $17.06 \times$ to $2.84 \times$ 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. 3. 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 memory bandwidth (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 memory bandwidth, 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, which in turn makes it more sensitive to its allocated cache space.

We repeated the experiment with each PARSEC benchmark to examine the effect of the workload characteristics. Our results show that the above observed pattern varies across benchmarks.

Observation 2. The relation between a task’s WCET and its allocated cache and memory bandwidth resources varies across different benchmark tasks. Some tasks (e.g., canneal benchmark) are sensitive to both cache and memory bandwidth 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 memory bandwidth resources to achieve better utilization and schedulability. In the next section, we formally define our resource allocation problem in this setting.

4 Theoretical modeling and problem statement

Using our prototype, we can already partition cache and memory bandwidth 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 memory bandwidth resources among cores, so as to minimize resources 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 that we observed from our empirical evaluation.

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 memory bandwidth partitions. Cache and memory bandwidth allocation is done at the core level: each core is allocated a distinct set of cache partitions and a certain number of memory bandwidth 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 bandwidth partitions that a core must be allocated, respectively. We assume that the OS scheduler schedules tasks on the cores using the partitioned Earliest Deadline First (pEDF) scheduling policy. However, our strategy will work with other partitioned scheduling policies as well (simply by replacing EDF schedulability test with the appropriate schedulability test).

**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 memory bandwidth allocation. Specifically, a task $\tau_i$ is modeled as $\tau_i = \{p_i, d_i, e_i(cp_i, bw_i) \mid N_{cp}^{min} \leq cp_i \leq N_{cp}, N_{bw}^{min} \leq bw_i \leq N_{bw}\}$, 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. We assume that all task parameters are given a priori. (Here, the vector of WCETs $e_i(cp_i, bw_i)$ of a task $\tau_i$ can be obtained by analysis or measurement. In our evaluation, it is obtained through profiling, as was done in our experimental evaluation.) As in [23, 49], we assume that a task’s WCET values have been inflated to account for other sources of overhead (e.g., cache-related preemption delay among tasks on the same core, extra memory access delay from cores with limited memory bandwidth, overhead introduced by other hardware resources such as the TLB, and interferences introduced by system software).$^4$

For analysis purpose, we refer to $re_i = e_i(N_{cp}, N_{bw})$ as the reference WCET of $\tau_i$, 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 $\tau_i$, 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 [29], 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 resource allocation strategy for computing (i) a mapping of tasks to cores, and (ii) the number of cache partitions and the number of memory bandwidth partitions (per regulation period) for each core in the system, so that the system is schedulable and the number of cores needed is minimized.

## 5 Optimal resource allocation algorithm

In this section, we present a Mixed Integer Programming (MIP) formulation of the resource allocation problem, which can be solved optimally using existing constraint solvers. Such an optimal solution is useful in providing a reference for evaluating the effectiveness and efficiency of our resource allocation strategy, which we present in the following section.

First, observe that on a platform with cache and memory bandwidth isolation, using more cores does not always lead to better schedulability. This is because when more cores are active, each core also has fewer cache and memory partitions on average. Hence, even though there are fewer tasks per core on average, the WCET of a task will also become larger; hence, it may become harder to meet the task’s deadline. Due to this behavior, to find the minimum number of cores to feasibly schedule a taskset, we need to perform schedulability analysis for each valid number of cores $m$, where $1 \leq m \leq M$ and $M$ is the maximum number of cores supported by the hardware.

---

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

$^4$ It would be interesting to incorporate existing overhead analysis such as [25, 39, 51] to explicitly account for these extra overheads in our resource allocation algorithm.
We now describe the constraints of our MIL formulation and its optimization goal. All notations for the formulation are summarized in Table 1.

**Table 1** Notations: constants are in upper case and variables are in lower case, by default.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{cp}$</td>
<td>The maximum number of cache partitions supported by the hardware</td>
</tr>
<tr>
<td>$N_{bw}$</td>
<td>The maximum number of bandwidth partitions supported by the hardware</td>
</tr>
<tr>
<td>$N_{min}^{cp}$</td>
<td>The minimum number of cache partitions per core</td>
</tr>
<tr>
<td>$N_{min}^{bw}$</td>
<td>The minimum number of bandwidth partitions per core</td>
</tr>
<tr>
<td>$N$</td>
<td>The total number of tasks in the system</td>
</tr>
<tr>
<td>$m$</td>
<td>The number of activated physical cores, which is a constant in the MIP problem</td>
</tr>
<tr>
<td>$\phi^{cp}_p$</td>
<td>The number of cache partitions allocated to core $p$</td>
</tr>
<tr>
<td>$\phi^{bw}_p$</td>
<td>The number of memory bandwidth partitions allocated to core $p$</td>
</tr>
<tr>
<td>$\chi(p)$</td>
<td>Binary variable to decide if task $i$ is assigned to core $p$</td>
</tr>
<tr>
<td>$\chi(cp,bw)$</td>
<td>Binary variable to decide if $i$ is allocated $cp$ cache partitions and $bw$ memory bandwidth partitions, where $cp$ and $bw$ are integers</td>
</tr>
</tbody>
</table>

The first two constraints (C1–C2) specify the constraints on the minimum resource that each core must have and the maximum resource that all cores can have. The next four constraints (C3–C6) enforce that every task must be assigned to a core, and that the task’s assigned WCET is determined by the cache and memory bandwidth given to it. The last constraint (C7) specifies the schedulability condition on each core. This constraint ensures that the system is schedulable if a set of values for the variables that satisfy these constraints exist, the optimization goal simply is to find a feasible solution.

**Constraint C1.** The total number of cache (memory bandwidth) partitions allocated to all activated cores must be no more than the total number of partitions of the platform.

\[
\sum_{0 \leq p < m} \phi^{cp}_p \leq N_{cp} \tag{1}
\]

\[
\sum_{0 \leq p < m} \phi^{bw}_p \leq N_{bw} \tag{2}
\]

**Constraint C2.** The number of partitions allocated to each core $p$ must be at least equal to the minimum number of the partitions the hardware enforces per core for each resource type.

\[
N_{min}^{cp} \leq \phi^{cp}_p, \quad \forall 0 \leq p < m \\
N_{min}^{bw} \leq \phi^{bw}_p, \quad \forall 0 \leq p < m \tag{3}
\]

**Constraint C3.** Each task must be assigned to exactly one core.

\[
\sum_{0 \leq p < m} \chi(p) = 1, \quad \forall 0 \leq i < N \tag{4}
\]

**Constraint C4.** Exactly one WCET value must be selected for each task.

\[
\sum_{N_{min}^{cp} \leq cp \leq N_{cp}} \sum_{N_{min}^{bw} \leq bw \leq N_{bw}} \chi(cp,bw) = 1, \quad \forall 0 \leq i < N \tag{5}
\]

**Constraint C5.** The assigned WCET ($e_i$), the assigned number of cache partitions ($cp_i$), and the
assigned number of bandwidth partitions \((bw_i)\) of each task \(\tau_i\) are determined as follows:

\[
e_i = \sum_{\text{N}_{\text{cp}}^\text{min} \leq cp \leq \text{N}_{\text{cp}}^\text{max}} \sum_{\text{N}_{\text{bw}}^\text{min} \leq bw \leq \text{N}_{\text{bw}}^\text{max}} \chi_i(cp, bw) \cdot e_i(cp, bw)
\]

\[
cp_i = \sum_{\text{N}_{\text{cp}}^\text{min} \leq cp \leq \text{N}_{\text{cp}}^\text{max}} \sum_{\text{N}_{\text{bw}}^\text{min} \leq bw \leq \text{N}_{\text{bw}}^\text{max}} \chi_i(cp, bw) \cdot cp
\]

\[
bw_i = \sum_{\text{N}_{\text{cp}}^\text{min} \leq cp \leq \text{N}_{\text{cp}}^\text{max}} \sum_{\text{N}_{\text{bw}}^\text{min} \leq bw \leq \text{N}_{\text{bw}}^\text{max}} \chi_i(cp, bw) \cdot bw
\]

**Constraint C6.** The number of partitions assigned to each task \(\tau_i\) must be equal to the number of partitions assigned to its core, for each type of resources.

\[
cp_i = \sum_{0 \leq p < m} \varpi_i(p) \cdot \varphi_p^{\text{cp}}, \quad \forall 0 \leq i < N
\]

\[
bw_i = \sum_{0 \leq p < m} \varpi_i(p) \cdot \varphi_p^{\text{bw}}, \quad \forall 0 \leq i < N
\]

**Constraint C7.** The total utilization of the tasks on each core \(p\) must be no larger than 1.

\[
\sum_{0 \leq i < N} \varpi_i(p) \cdot \frac{e_i}{p_i} \leq 1, \quad \forall 0 \leq p < m
\]

**Objective.** Minimize \(m\).

By construction, if there exists at least one feasible configuration to schedule the taskset on \(m\) cores, the solver will eventually find a feasible configuration for resource allocation and task mapping. We used the Gurobi solver [6] to obtain a solution to the MIP problem for each value \(m\) between 1 and the maximum number of cores supported by the platform. One can easily verify that the smallest \(m\) for which a solution exists is the minimum number of cores required to schedule the taskset, and the corresponding solution provides the task mapping and resource allocation for cores.

Although the MIP-based solution is optimal, it has an exponential time complexity, which makes it impractical in practice. In the next section, we present an alternative solution that is much more efficient while also effective, using a combination of clustering and bin-packing heuristics.

### 6 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.

#### 6.1 Overview

Based on the insights obtained from the empirical evaluation in Section 3.3, 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 memory bandwidth resources (equal to that of the core), grouping tasks with similar sensitivity to the cache and memory bandwidth resource allocations onto the same core can help better utilize the cache and memory bandwidth resources.
Towards this, we define the resource-allocation slowdown of a task $t_i$ under $cp_i$ cache partitions and $bw_i$ memory bandwidth partitions to be $\tau_{\text{slowdown}}(cp_i, bw_i) = \frac{s_i(cp_i, bw_i)}{re_i}$, where $re_i$ is the task’s reference WCET (i.e., the WCET when the task is given all partitions; c.f. Section 4.) Our allocation strategy aims to group tasks with similar resource-allocation slowdown onto the same core.

The next strategy simply aims to balance load across cores:

**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 a core $i$’s resource utility as the average reduced utilization per newly allocated cache and memory bandwidth partition for the core:

$$resourceU_i(cp, bw) = \begin{cases} \frac{(u_i - u'_i)}{(cp + bw)} & \text{if } u_i > 1 \\ 0 & \text{otherwise} \end{cases}$$

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. To balance the load across cores, we will find an allocation that maximizes the resource utility whenever assigning some extra partitions to a core:

**Strategy 3.** When adding more cache and memory bandwidth 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 memory bandwidth 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 memory bandwidth partitions. The algorithm then works in three phases:

1. **Phase 1** (Lines 1–2): It first groups tasks that have similar sensitivity to cache and memory bandwidth 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 memory bandwidth 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
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. $clusters \leftarrow clusterTasks(V, m, maxIterKM)$
2. Sort tasks in each cluster in decreasing order of their reference utilization
3. repeat
4. $perm\_clusters \leftarrow permute(clusters)$ ▷ randomly pick one permutation of $clusters$
5. $cores \leftarrow binPackClusters(perm\_clusters, m)$
6. $cores \leftarrow allocResource(cores, m, N_{cp}, N_{bw})$ ▷ The $cores$ variable specifies tasks and resources allocated to each core
7. $sched \leftarrow checkSchedulability(cores)$ ▷ schedulable if each core’s assigned utilization is no larger than 1
8. if $sched =$ schedulable then
9. break
10. $oldVal \leftarrow \infty$ ▷ previous imbalance value
11. while true do ▷ balance $cores$’ utilizations iteratively
12. $val \leftarrow getImbalanceValue(cores)$
13. $cores \leftarrow balance(cores)$
14. $cores \leftarrow allocResource(cores, m, N_{cp}, N_{bw})$
15. $sched \leftarrow checkSchedulability(cores)$
16. if $sched =$ schedulable or $val > oldVal$ then
17. break
18. $oldVal \leftarrow val$
19. $maxIterPerm \leftarrow maxIterPerm - 1$
20. until $maxIterPerm = 0$
21. return $sched$
22.
23. function getImbalanceValue(cores)
24. $imbalance = 0$
25. for all $c \in cores$ do
26. if $c$’s assigned utilization $> 1$ then
27. $imbalance += c$’s assigned utilization $- 1$
28. return $imbalance$ rounded to 2 fractional digits
memory bandwidth resources available to each core on average (since each active core must have at least some cache and memory bandwidth 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.

Algorithm 2  
\textbf{Algorithm 2} \; \text{clusterTasks}(V, m, maxIterKM)  
\textbf{Input:} \; \tau: \text{the taskset}, m: \text{the number of cores}, maxIterKM: \text{the maximum iterations.}  
\textbf{Output:} \; m \text{ clusters of tasks}  
\begin{enumerate}
\item Calculate each task’s resource-allocation slowdowns  
\item Create \( m \) clusters \( C \) with \( m \) randomly picked tasks as its centroid  
\item \textbf{repeat}  
\item \textbf{for all} task \( \tau_i \in \tau \) do  
\item \text{min\_distance} \leftarrow \infty  
\item \textbf{for all} cluster \( c \in C \) do  
\item \text{if} \( \text{dist}(\tau_i, c) < \text{min\_distance} \) \;& \; \text{\texttt{dist}}(\tau_i, c) \text{ is distance between } \tau_i \text{ and } c.  
\item \text{cluster} \leftarrow c; \; \text{min\_distance} \leftarrow \text{dist}(\tau_i, c)  
\item \text{if} \( \tau_i \notin \text{cluster} \) then  
\item \text{updated} \leftarrow \text{true}; \; \text{cluster} \leftarrow \tau_i  
\item \text{end} \; \text{if}  
\item \text{end} \; \text{for all}  
\item \text{end} \; \textbf{repeat}  
\item \textbf{for all} cluster \( c \in C \) do  
\item Calculate the mean of all tasks in cluster \( c \)  
\item Update the new mean as cluster \( c \)’s new centroid  
\item \textbf{until} \( \text{updated} = \text{false} \) \textbf{or} \( \text{maxIterKM} = 0 \)  
\end{enumerate}

Algorithm 3  
\textbf{Algorithm 3} \; \text{binPackClusters}(C, m)  
\textbf{Input:} \; C: \text{m clusters of tasks}, m: \text{number of cores}  
\textbf{Output:} \; P: \text{m cores that are packed with tasks and have similar reference utilizations}  
\begin{enumerate}
\item Initialize \( P \) as \( m \) empty cores; \; \text{sumRU} \leftarrow 0  
\item \textbf{for all} cluster \( c \in C \) do  
\item \text{refUtil} \leftarrow c; \; \text{sumRU} \leftarrow \text{refUtil} \quad \text{\texttt{refUtil} is cluster } c \text{’s reference utilization}  
\item \text{meanRefU} \leftarrow \text{sumRU} / m  
\item \textbf{for all} cluster \( c \in C \) do  
\item \textbf{for all} task \( \tau_i \in c \) do  
\item \text{chosen} \leftarrow P_j; \; tU_j \leftarrow \tau_i \quad \text{\texttt{tU}_j \text{ is task } \tau_i \text{’s reference utilization}}  
\item \textbf{for} \; j = 0; j < m; j = j + 1 \textbf{do}  
\item \text{P}_j \leftarrow P_j; \; pU_j \leftarrow P_j \quad \text{\texttt{P}_j \text{ is the } j^{th} \text{ core in } P, pU_j \text{ is } P_j \text{’s reference utilization}}  
\item \text{if} \; pU_j < \text{meanRefU} \quad \text{\texttt{pU}_j \text{ is } P_j \text{’s reference utilization}}  
\item \text{chosen} \leftarrow P_j; \; \text{break} \quad \text{\texttt{Find a candidate core to pack the task } \tau_i}  
\item \text{end if}  
\item \text{end for}  
\item \text{chosen} \leftarrow \tau_i \quad \text{\texttt{Assign task } \tau_i \text{ to chosen core}}  
\item \text{end for}  
\item \text{end for}  
\item \text{return} \; P  
\end{enumerate}
Algorithm 4 allocResource($C, m, N_{cp}, N_{bw}$)

**Input**: $C$: $m$ cores, $N_{cp}$: the number of available cache partitions, $N_{bw}$: the number of available memory bandwidth partitions

**Output**: The number of cache and memory bandwidth partitions allocated for each core that maximizes the resource utility

1: for all core $c \in C$ do
2:   $c \leftarrow (N_{cp}^{\min} \cdot N_{bw}^{\min})$ \hspace{1em}$\triangleright$ Assign minimum cache partitions and bandwidth partitions to $c$
3:   $N_{cp}^{idle} = N_{cp} - m \cdot N_{cp}^{\min}$, $N_{bw}^{idle} = N_{bw} - m \cdot N_{bw}^{\min}$
4: while $N_{cp}^{idle} > 0$ or $N_{bw}^{idle} > 0$ do
5:   $\maxRU = 0$ \hspace{1em}$\triangleright$ Max resource utility
6:   for all core $c \in C$ do
7:     /* Get the optimal resource utility $cMaxRU$ of core $c$ when the core is given $cChosenCP$ cache and $cChosenBW$ memory bandwidth partitions by solving Eq. 10*/
8:     $(cMaxRU, cChosenCP, cChosenBW) \leftarrow getMaxResUtility(c, N_{cp}^{idle}, N_{bw}^{idle})$
9:     if $cMaxRU > \maxRU$ then
10:        $\maxRU \leftarrow cMaxRU$; $\text{chosen}CP \leftarrow cChosenCP$;
11:        $\text{chosen} \leftarrow c$; $\text{chosenBW} \leftarrow cChosenBW$;
12:        if $\text{chosenCP} = 0$ and $\text{chosenBW} = 0$ then
13:            break \hspace{1em}$\triangleright$ Stop due to no schedulability benefit
14:     /* Allocate found resource to core $\text{chosen}$ */
15:     $\text{remainCP} -= \text{chosenCP}$; $\text{remainingBW} -= \text{chosenBW}$
16: return $C$

Algorithm 5 balance($C$)

**Input**: $C$: $m$ cores whose number of cache and bandwidth partitions is determined

**Output**: Cores whose assigned utilizations are balanced

1: Create $uC$ which holds all unschedulable cores in $C$
2: Sort $uC$ in decreasing order of cores’ assigned utilizations
3: Sort tasks in each core in increasing order of a task’s assigned slowdown (i.e., assigned utilization / reference utilization)
4: for all unschedulable core $uc \in uC$ do
5:   for all task $\tau_i \in uc$ do
6:      $\text{curU} \leftarrow \tau_i$ \hspace{1em}$\triangleright$ $\tau_i$’s assigned utilization in $uc$
7: /* Find a core that has smallest assigned utilization after $\tau_i$ is moved to that core */
8:   for $c \in C$ do
9:      if $c = c$ then
10:         continue
11:      $\text{util} \leftarrow c \cup \tau_i$ \hspace{1em}$\triangleright$ $c$’s assigned utilization if $\tau_i$ moves to $c$
12:      if $\text{util} < \text{min_util}$ then
13:         $\text{min_util} \leftarrow \text{util}$; $\text{dst} \leftarrow c$ \hspace{1em}$\triangleright$ Move $\tau_i$ to $\text{dst}$ core
14:      Update assigned utilization for $uc$ and $\text{dst}$ cores
15:      if $uc$’s assigned utilization $\leq 1$ then
16:         break
17: return $C$
6.2 Details of the algorithm

We now discuss the key ideas of the four main procedures used in our algorithm: clusterTasks(), binPackClusters(), allocResource(), and balance(). Their pseudo-code is shown in Algorithms 2–5.

Algorithm 2: clusterTasks(). This procedure uses the KMeans algorithm [31]—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. Each task \( \tau_i \) has a \( N_{\text{config}} \) dimensional slowdown vector \( \tilde{s}_{ij} \), where \( N_{\text{config}} \) is the number of valid resource configurations. (A valid resource configuration is a pair of a valid number of cache partitions and a valid number of memory bandwidth partitions.) The \( \tilde{F} \) element of the slowdown vector is the resource allocation slowdown of the task under the corresponding resource configuration. Formally, the procedure aims to divide the set of tasks \( \tau \) into \( m \) clusters, such that the pairwise deviation of tasks in the same cluster is minimized:

\[
\arg\min_{C} \sum_{k=1}^{m} \frac{1}{2|C_k|} \sum_{\tau_j, \tau_i \in C_k} ||\tilde{s}_{ij} - \tilde{s}_{ij}||^2 \tag{9}
\]

where \(|C_k|\) is the number of tasks in the cluster \( C_k \).

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.

Algorithm 3: binPackClusters(). This 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 \( \text{meanRefU} \) 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 \( \text{meanRefU} \) 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.

Algorithm 4: allocResource(). This procedure allocates cache and memory bandwidth resources to cores, such that the system is schedulable while minimizing cache and memory bandwidth resources. We define the optimal resource utility of a core \( c_i \) when the system has \( N_{cp}^{\text{idle}} \) unused cache partitions and \( N_{bw}^{\text{idle}} \) 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 \( cp \leq N_{cp}^{\text{idle}} \) and \( bw \leq N_{bw}^{\text{idle}} \). That is,

\[
\text{optResourceU}_i(N_{cp}^{\text{idle}}, N_{bw}^{\text{idle}}) = \max_{N_{cp}^{\text{idle}} \leq cp \leq N_{cp}^{\text{idle}}, N_{bw}^{\text{idle}} \leq bw \leq N_{bw}^{\text{idle}}} \text{resourceU}_i(cp, bw) \tag{10}
\]

The allocResource() procedure has three steps: (1) Initialization step, which allocates the minimum number of cache and memory bandwidth partitions to each core. (2) Resource allocation step, which allocates \( cp \) cache partitions and \( bw \) memory bandwidth partitions to the core \( c \) that has the maximum optimal resource utility with \( N_{cp}^{\text{idle}} \) cache partitions and \( N_{bw}^{\text{idle}} \) memory bandwidth partitions available. In other words, it follows Eq. (11) to allocate some or all remaining partitions to the core that has the maximum optimal resource utility:

\[
\arg \max_{cp, bw, c \in C} \text{optResourceU}_i(N_{cp}^{\text{idle}}, N_{bw}^{\text{idle}}) \tag{11}
\]
Holistic resource allocation for multicore real-time systems

where $C$ is the set of cores, and we will allocate $cp$ cache partitions, $bw$ memory bandwidth partitions to the core $c$ with the currently remaining $N_{idle}^{c}$ idle cache partitions and $N_{idle}^{bw}$ idle memory bandwidth partitions. (3) Update step, which updates the remaining amount of cache and memory bandwidth resources and the utilization of the core $c$. Finally, the procedure repeats until all cores become schedulable or there is no benefit in reducing the utilization of an unschedulable core.

Algorithm 5: balance(). This 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 sub-procedures (i.e., Algorithms 2–5). Algorithm 2, $clusterTasks()$, enumerates all clusters for all tasks for $maxIterKM$ iterations; hence, it takes $O(N \cdot m \cdot maxIterKM)$ time. Algorithm 3, $binPackCluster()$, iterates over all tasks in each cluster (i.e., $N$ tasks in total) for each core, which takes $O(N \cdot m)$ time. Algorithm 4, $allocResource()$, takes $O(N_{cp}, N_{bw})$ to calculate Eq 10 for each core and iterates for maximum $N_{cp}, N_{bw}$ times, thus it takes $O(m \cdot N_{cp}^2 \cdot N_{bw}^2)$ time. Algorithm 5, $balance()$, sorts at most $N$ tasks and iterates at most $N$ tasks for at most $m$ cores, so it takes $O(N \cdot \log N) + O(N \cdot m)$ 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 Algorithms 5 and 4 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 Algorithms 3 and 4 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: $O(N \cdot m \cdot maxIterKM) + O(maxIterPerm \cdot max\{N \cdot m \cdot \log N, m \cdot N_{cp}^2 \cdot N_{bw}^2\})$.

7 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 four other solutions: the optimal algorithm in Section 5, and three baseline algorithms that combine existing bin-packing heuristics with evenly distributing cache and memory bandwidth resources to cores.

7.1 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 [9]. A task’s WCET values under different cache and memory bandwidth 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 three types of inputs (simsmall, simmedium, and simlarge) under different cache and memory bandwidth configurations using CaM prototype on the machine we used for our empirical evaluation (c.f. Section 3). For each PARSEC benchmark with each type of input, we dedicated one core for the benchmark, configured the core with a valid cache and bandwidth 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 bandwidth partitions was ranged from 1 to 20, with a step of 1. The set of valid cache and bandwidth configurations is a cartesian product of the valid number of cache partitions and the valid number of memory bandwidth partitions. For each PARSEC benchmark with each type of input, we measured its WCETs under $19 \times 20 = 380$ valid cache and memory bandwidth 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 that we have available: Platform A has 4 cores and 20 cache partitions; Platform B has 4 cores and 12 cache partitions. The number of memory bandwidth partitions is the same as the number of cache partitions on each platform.

**Baseline algorithms.** The baseline algorithms evenly distribute cache and memory bandwidth resources to cores. Each core has $N_{cp}/M$ cache partitions and $N_{bw}/M$ memory bandwidth partitions, where $N_{cp}$ is the total number of cache partitions, $N_{bw}$ is the total number of memory bandwidth 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 algorithms use 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. For the baseline algorithms, we considered three bin-packing approaches: first-fit (Baseline-FF), best-fit (Baseline-BF), and worst-fit (Baseline-WF).

**Analysis.** We analyzed the same set of tasksets with each of the five algorithms: our algorithm (Heuristic), the optimal algorithm (Optimal) and three baseline algorithms (Baseline-FF, Baseline-BF, Baseline-WF). For our algorithm, we considered three settings of $(\text{maxIterKM}, \text{maxIterPerm})$, including $(100, 24)$, $(50, 12)$, and $(200, 48)$. As the performance results are consistent across these settings, we present only the results for $(100, 24)$.

Our analyses were performed on an Intel Xeon E5-2683 v4 processor, which has 32 cores (with hyper threading enabled) operating at 2.10GHz. We set the analysis timeout value as 8 hours\(^5\). We refer to a taskset for which an analysis timed out as an incomputable taskset for the analysis.

### 7.2 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 25 independent tasksets (i.e., 775 tasksets in total), with tasks’ utilizations uniformly distributed in $[0.1, 0.4]$. We analyzed the tasksets for Platform A using the five algorithms. Fig. 4 shows the fraction of schedulable tasksets under each algorithm.

---

\(^5\) 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.
The results show that the fraction of schedulable tasksets under our heuristic algorithm was very close to that of the optimal algorithm. Only 2.58% of the tasksets (20 out of 775) were deemed schedulable by the optimal solution but deemed unschedulable under our heuristic algorithm. Further, we observe that our heuristic algorithm significantly outperformed the best baseline algorithm (Baseline-BF): our heuristic algorithm was able to schedule 58% more schedulable tasksets (627/389 = 61.18%) compared to the baseline algorithm.

7.3 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 five algorithms. For this, we repeated the above experiment on another platform (i.e., Platform B) and reported the results in Fig. 5. 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 487 tasks, while our heuristic algorithm could schedule 467 schedulable tasksets, i.e., 1 – 467/487 = 4.11% fewer than under the optimal algorithm. Our heuristic algorithm also substantially outperformed the baseline algorithms; for instance, the best baseline algorithm (Baseline-BF) could schedule 219 tasksets, which is 467/219 – 1 = 113.24% fewer than that of our algorithm.

Impact of task parameters. We investigated the impact of the task parameters on the fraction of schedulable tasksets for all five algorithms. We repeated the experiment in Section 7.2 using tasksets with uniform-heavy and uniform-light utilization distributions. The results are shown in Fig. 6.
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 516 schedulable tasksets, while our heuristic algorithm had 502 schedulable tasksets, which is only \( \frac{502}{516} = 2.71\% \) fewer.

We also 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 2.0, while our heuristic algorithm never timed out. Table 2 summarizes the number of schedulable tasksets, unschedulable tasksets and incomputable tasksets for all five 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 the optimal solution: at most 15.35\% of the tasksets, i.e., \((410 + 343 - 634)\) out of 775, would be deemed schedulable by the optimal solution but unschedulable under our heuristic algorithm. It can also be observed here that our heuristic algorithm outperformed the baseline algorithm by a significant factor. 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.

### Table 2 Number of analyzed tasksets (out of 775 tasksets) for uniform-light tasks on Platform A.

<table>
<thead>
<tr>
<th></th>
<th>Heuristic</th>
<th>Optimal</th>
<th>Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FF</td>
<td>BF</td>
<td>WF</td>
</tr>
<tr>
<td>Schedulable</td>
<td>634</td>
<td>343</td>
<td>441</td>
</tr>
<tr>
<td>Unschedulable</td>
<td>141</td>
<td>22</td>
<td>334</td>
</tr>
<tr>
<td>Incomputable</td>
<td>0</td>
<td>410</td>
<td>0</td>
</tr>
</tbody>
</table>

#### 7.4 Running time efficiency

We measured the running time of our algorithm and the optimal algorithm in the evaluation in Fig. 6b. We observed that our algorithm is highly efficient: its average running time was 0.80 minutes. On the contrary, the optimal algorithm’s running time increased exponentially as the taskset’s utilization increased, and its average running time was 400 minutes, which is \( \frac{400}{0.80} = 500 \times \) slower than our heuristic algorithm.
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 3). 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 a baseline (vanilla) that uses first-fit bin-packing algorithm to pack tasks onto cores in LITMUS\textsuperscript{RT} (without cache or memory bandwidth management support).

**Workload.** We first converted the PARSEC benchmarks into LITMUS\textsuperscript{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 3) as a background task.

**Experiment.** Under the heuristic and optimal settings, we reserved one core, two cache partitions and one memory bandwidth 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. 8 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. 8, 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 in our experiment.

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.

Conclusion

We have presented a resource allocation strategy for hard 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 memory bandwidth 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.
References


Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yu Peng Li, Zhihao Yu, Bowen Huang, Tianxi Xu, Zhicheng Yao, Yun Chen, Haibin Wang, Lixin Zhang, and Yungang Bao. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (PARD). In *ASPLOS*, 2015.


Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS*, 2012.


