Holistic Resource Allocation under Federated Scheduling for Parallel Real-Time Tasks

LANSHUN NIE* and CHENGHAO FAN*, Harbin Institute of Technology, CHN
SHUANG LIN, Harbin Institute of Technology, CHN
LI ZHANG, Amazon Web Services, USA
YAJUAN LI, New Jersey Institute of Technology, USA
JING LI†, New Jersey Institute of Technology, USA

With the technology trend of hardware and workload consolidation for embedded systems and the rapid development of edge computing, there have been increasing interests in supporting parallel real-time tasks to better utilize the multi-core platforms while meeting the stringent real-time constraints. For parallel real-time tasks, the federated scheduling paradigm, which assigns each parallel task a set of dedicated cores, achieves good theoretical bounds by ensuring exclusive use of processing resources to reduce interferences. However, because cores share the last-level cache and memory bandwidth resources, in practice tasks may still interfere with each other despite executing on dedicated cores. Such resource interferences due to concurrent accesses can be even more severe for embedded platforms or edge servers, where the computing power and cache/memory space are limited. To tackle this issue, in this work, we present a holistic resource allocation framework for parallel real-time tasks under federated scheduling. Under our proposed framework, in addition to dedicated cores, each parallel task is also assigned with dedicated cache and memory bandwidth resources. We study the characteristics of parallel tasks upon different resource allocations following a measurement-based approach and proposes a technique to handle the challenge of tremendous profiling for all resource allocation combinations under this approach. Further, we propose a holistic resource allocation algorithm that well balances the allocation between different resources to achieve good schedulability. Additionally, we provide a full implementation of our framework by extending the federated scheduling system with Intel's Cache Allocation Technology and MemGuard. Finally, we demonstrate the practicality of our proposed framework via extensive numerical evaluations and empirical experiments using real benchmark programs.

CCS Concepts: • **Computer systems organization** → **Real-time system specification**; **Embedded software**; *Embedded systems*; • **Software and its engineering** → **Real-time schedulability**; • **Theory of computation** → Parallel computing models.

Additional Key Words and Phrases: parallel real-time systems, federated scheduling, resource partitioning

ACM Reference Format:
Lanshun Nie, Chenghao Fan, Shuang Lin, Li Zhang, Yajuan Li, and Jing Li. 2022. Holistic Resource Allocation under Federated Scheduling for Parallel Real-Time Tasks. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2022), 29 pages. https://doi.org/10.1145/3489467

*Both authors contributed equally to this research.
†Jing Li is the corresponding author.

Authors' addresses: Lanshun Nie; Chenghao Fan, Harbin Institute of Technology, CHN; Shuang Lin, Harbin Institute of Technology, CHN; Li Zhang, Amazon Web Services, USA; Yajuan Li, New Jersey Institute of Technology, USA; Jing Li, New Jersey Institute of Technology, USA.

1539-9087/2022/1-ART1 $15.00
https://doi.org/10.1145/3489467

1 INTRODUCTION

Because workload consolidation can effectively reduce the energy consumption, wiring weight, hardware costs, and software complexity, recently there is a technology trend of consolidating even more applications and services onto shared hardware for embedded systems and edge servers. On the hardware side, there is a rapidly increasing penetration of multi-core/many-core CPUs into systems, as well as an increased sharing of common resources by computing units (e.g., memory bus and last-level cache). Moreover, the applications running in embedded systems and edge servers today have increasingly high computation needs and stringent timing constraints. For example, an edge server needs to provide real-time responsiveness to various applications, such as augmented reality, video analytics and traffic light controls, that offload computation to the shared edge [40, 41]. These technology trends mean that: (1) the parallel execution is critical for satisfying the high computation needs and meeting the stringent real-time constraints of applications; and (2) the execution of applications is more unpredictable due to computing units sharing resources like cache and memory bus.

Among the different scheduling policies for executing parallel real-time tasks upon multi-core platforms, the federated scheduling paradigm [2, 7, 8, 21, 28, 43, 44] has attracted a lot of attention due to its good theoretical bounds and empirical advantages. The key idea of the federated scheduling paradigm is to allocate a set of dedicated cores to each task that needs to run in parallel on multiple cores to meet its deadline and force the remaining tasks to execute serially on the remaining cores under some classical multiprocessor scheduling algorithms. Because each task that runs in parallel has its dedicated cores for execution, there is no preemption, migration, and interference on its cores caused by other tasks. In addition to reducing practical overheads, each of such tasks can be analyzed in isolation, which significantly reduces the pessimism of analyzing the schedulability of complex parallel real-time tasks. Thus, federated scheduling achieves the best performance bounds compared to other classical algorithms, such as global earliest deadline first and global rate monotonic scheduling.

However, in today's multicore hardware, cores share the last-level cache and memory bandwidth resources, so tasks may interfere with each other despite executing on dedicated cores. The interferences due to cache and memory bandwidth contention can be even more severe for embedded platforms or edge servers, where the computing power and cache/memory space are limited.

This work aims to address the pressing demand for *parallel real-time scheduling over multicore platforms with shared cache and memory bus*. Specifically, we focus on platforms with multicore processors with L1 (and L2) caches private to each core and shared Last Level Cache (LLC), which is connected via a shared memory bus to the shared Direct Random Access Memory (DRAM). In this work, we take a measurement-based approach and limit our attention to addressing the contention in the shared LLC and memory bus for parallel real-time tasks. Note that there are other sources of interference on modern platforms, such as Miss Status Holding Registers (MSHRs) [45], DRAM bank conflicts [23, 51], and DRAM controller [37, 42], as well as the contention in software systems like blocking due to shared internal kernel data structures [56]. Some of these interferences (e.g., MSHRs contention) can be mostly incorporated into the measured worst-case execution times by co-running the task with specially designed interfering workloads during the profiling, while some have been mitigated by various mechanisms. We leave the integration of the proposed strategy and the orthogonal mitigation mechanisms to other resources, such as DRAM bank-level partitioning, as future work.

We first study the characteristics of parallel tasks upon different resource allocations following a measurement-based approach. Since each task can be allocated with different numbers of cores, cache partitions, and memory bandwidth partitions, profiling the worst-case execution times for

all the thousands of combinations of resource allocation can take a tremendous amount of time. To address this issue, we propose to perform the measurement only for a small number of combinations and apply a non-linear regression to obtain estimations for the other combinations. Next, we present a holistic cache and memory bandwidth resource allocation strategy for parallel real-time tasks under federated scheduling. In addition to dedicated cores, each parallel task is also assigned with dedicated cache and memory bandwidth resources to reduce resource interferences between tasks. We leverage the insights from the heuristic resource allocation strategy CaM[48] for allocating cache and memory bandwidth for sequential tasks and extend the federated scheduling system using Intel's Cache Allocation Technology and MemGuard for allocating cache and memory bandwidth. To well balance the allocation between different resources and achieve good schedulability, we develop a mixed-integer nonlinear programming (MINLP) formulation that can optimally solve this problem. Moreover, we propose a heuristic-based greedy algorithm that has good schedulability and short running times that are orders of magnitude faster than solving the MINLP. Additionally, we provide a full implementation of our framework by extending the federated scheduling system with Intel's Cache Allocation Technology and MemGuard. Finally, we demonstrate the practicality of our proposed framework via extensive numerical evaluations and empirical experiments using real benchmark programs.

2 RELATED WORK

Parallel real-time scheduling. The problem of scheduling parallel real-time tasks has been broadly studied. The earlier works develop a task decomposition technique to apply the analysis of multiprocessor scheduling [22, 25, 26, 33, 38, 46]. For directly scheduling parallel tasks, classic schedulers [4, 10, 12, 27, 32, 34] and Federated Scheduling that is specifically designed for parallel tasks [2, 21, 28, 44] have been analyzed. All of them, except for [2, 34, 42, 43], only consider how to allocate cores to parallel tasks and do not consider the contention in cache and memory bandwidth. Alhammad and Pellizzoni, for the first time, analyze the memory bandwidth allocation for parallel tasks, using a theoretical approach. They model the memory time as part of the work and calculate a task's execution time given a certain number of cores and a certain amount of bandwidth assigned to the task. For analyzing private cache, a private-cache-aware algorithm is proposed for finding partitioned non-preemptive schedules for parallel tasks [34]. In contrast, we consider both shared cache and memory bandwidth for parallel tasks and takes an empirical approach based on measurements of WCET. E-WarP [42] is a framework that analyzes the fine-grained memory demand of applications and uses the developed memory enveloping to perform accurate WCET predictions under bandwidth regulation for both CPU and accelerator workload. To analyze the fine-grained cache behavior of parallel tasks, [43] incorporates the cache-aware BUNDLE-scheduling into federated scheduling for parallel tasks. Both works focus on analyzing the fine-grained behaviors of individual parallel tasks and improve their execution efficiencies. Thus, they are orthogonal to this work and can be integrated for better performance.

Allocating cache and memory bandwidth. Cache partitioning techniques, such as page coloring, have been studied extensively to reduce contention on cache [6, 13, 24, 54] (see [15] for a survey). Interference due to cache has also been incorporated into scheduler design and analysis [11, 16, 39, 47, 49]. Recent processors provide more efficient hardware support for cache partitioning [5, 20]. Analyses on memory controllers achieve deterministic memory access latency via detailed assumptions and/or modifications to controller hardware [17, 18, 23, 30, 55]. In contrast, software-based techniques regulate the memory bandwidth via throttling a core when the monitoring unit observes the excessive memory requests of the core [1, 50, 52]. CaM [48] proposes to holistically allocate cache and memory bandwidth to sequential tasks on multicore machines.

This idea is later incorporated into the compositional analysis for real-time virtualization to provide better timing isolation among tasks in VMs. To address shared cache and memory bus contention while ensuring task timing requirements in virtualized systems, Maracas [49] adopts page coloring techniques and a latency-based memory throttling approach. All the above research considers sequential tasks, while this work extends CaM for parallel tasks with federated scheduling.

3 IMPACT OF RESOURCE ALLOCATIONS ON PARALLEL REAL-TIME TASKS

To investigate the characteristics of the worst-case execution times of parallel real-time tasks when allocated with different amounts of cores, cache partitions, and memory bandwidth resources, in this section, we conduct an empirical evaluation using real-world parallel applications. Specifically, we extend parallel benchmark programs written in the widely used OpenMP [35] language using Intel CAT [20] and MemGuard [52] for dedicating resources in our experiments. The observations obtained from this empirical study not only motivate the importance of holistic resource allocation for parallel real-time tasks, but also stimulates us to apply a regression function on the measurement results. This regression function is later used to reduce the tremendous profiling effort for all different combinations of resource allocation in the measurement-based approach.

3.1 Experimental Setup

We first describe the resource allocation implementation and experimental setup for measuring the worst-case execution times of parallel benchmark programs upon different numbers of allocated cores, cache partitions, and memory bandwidth partitions.

CAT. Intel's Cache Allocation Technology (CAT) [20], which is available to Intel processors starting with the Xeon E5 v4 family, provides software-programmable control over the amount of last-level cache (LLC) that can be consumed by software or hardware threads. More specifically, CAT relies on mapping each running software or hardware thread onto an intermediate construct called a *Class of Service (CLOS)*. Then, CLOS can be configured via the L3 capacity bitmasks to set the available cache partitions, which associates the cache partitions with the software or hardware threads. Intel Resource Director Technology Software Package provides the OS interface leveraging Linux kernel extensions to achieve the assignment of cache partitions to a process (i.e., task) or a set of cores. In our system implementation, we configure the Linux kernel via `CONFIG_INTEL_RDT_A` to enable the two OS interfaces `pqos_l3ca_set` and `pqos_alloc_assoc_set` for allocating cache partitions. For a parallel task executed on multiple dedicated cores, we use these interfaces to assign cache partitions that are shared only by the cores of the task.

Memguard. Our implementation leverages the reservation mechanism of MemGuard [52] to allocate memory bandwidth to parallel tasks and cores. Specifically, MemGuard utilizes the hardware performance monitoring counter via the Linux `perf_event` infrastructure to monitor the last-level cache miss of each core. Since each cache miss generates a memory access request, one can calculate the maximumly allowed number of cache misses for a duration without exceeding the specified memory bandwidth. When reaching this number, MemGuard throttles the computation of this core by calling the `cpu_relax()`. At the end of the current duration, MemGuard resets the counter and wakes up the core for execution. In this way, MemGuard is able to restrict the amount of memory bandwidth used by each core. However, because the hardware counter can only monitor the cache misses for each core, MemGuard only supports individually allocating a certain amount of memory bandwidth to a core. But it does not allow allocating memory bandwidth that can be shared by a set of cores or by the multiple parallel threads a process on different cores. Hence, for a parallel task assigned with multiple dedicated cores, our implementation calculates the amount of memory bandwidth to be allocated for each of these cores by using the number of cores to divide the desired

total amount of memory bandwidth allocated to this task. Then, we use the interface provided by MemGuard to achieve this allocation.

Hardware. We conduct the experiments on a 14-core machine with an Intel Xeon Gold 5117 processor that supports Intel CAT. The cores in the processor share a 19.25MB L3 cache and 6-channel 32GB DDR4 DRAM with a maximum memory speed of 2400MHz. The shared L3 cache can be divided into 11 equal-size partitions. The processor has 8 Class of Service (CLOS) registers, so it supports at most 8 sets of cache partitions, where each set (i.e., each CLOS) must be assigned with at least one cache partition. We adopted the DRAM controller saturation analysis in [42] for obtaining the maximum memory bandwidth. For the workload with stores that always result in DRAM row misses, the maximum memory bandwidth without fully saturating the DRAM controller is 7.83 GB/s. In comparison, for the read-intensive workload that always results in cache misses, the maximum memory bandwidth without fully saturating the DRAM controller is 17.97 GB/s. We observe that the realistic benchmark programs described below typically generate more loads than stores. Hence, we consider a maximum guaranteed memory bandwidth of 12GB/s assuming a ratio of roughly two stores and one load. To discretize the amount of memory bandwidth that can be allocated to tasks, we divide the bandwidth into 20 partitions of 600MB/s each, where each task is assigned with one or multiple partitions. This number is chosen considering the balance between the sufficient number of partitions for the allocation and the sufficient size of each partition.

System configuration. Our experiments are run on Linux 4.15.0, where hyper-threading, Speed-Step, and hardware cache prefetcher features are disabled to reduce the non-determinism in the timing behavior of tasks. For both the benchmark profiling described here and the empirical evaluation of our framework in Section 8, we run the benchmark programs under the Linux real-time priorities. Note that Linux comes with a safeguard mechanism that throttles the execution under real-time priorities when reaching 95% CPU utilization by default. We disable the real-time scheduler throttling by setting `sched_rt_runtime_us` to -1. We further reserve one core (i.e., core 0) for system services, dedicated one cache partition to this core using CAT, and restrict its memory bandwidth usage using Memguard to limit the interference from system services to the experiments. In summary, we leave 13 cores, 10 cache partitions, and 20 memory bandwidth partitions for running experiments.

Parallel runtime system. We use GCC 7.4.0 with OpenMP 2.0 as the compiler and runtime system for executing parallel benchmarks. We configure OpenMP to generate and pin exactly one thread per core using `omp_set_num_threads()` and `pthread_setaffinity_np()`, so each parallel task uses and only uses its dedicated cores without thread migrations. To further reduce the variation of parallel execution times, we set `OMP_WAIT_POLICY` as active, disallow nested parallelism using `omp_set_nested(0)`, and set `GOMP_SPINCOUNT` as infinity.

Workload. We modified 12 parallel benchmark programs to enable the allocation of dedicated cores, cache partitions, and memory bandwidth partitions using the aforementioned interfaces provided by OpenMP, Intel CAT, and MemGuard, respectively. The 12 benchmark programs are converted from two widely used parallel benchmark suites that collect real-world applications with various parallel structures and properties. Specifically, `Facesim`, `Bodytrack`, `Fluid Animate` (`Fluid`), `Swaptions`, and `Blackscholes` are from the Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suites [9]; While `Ray Casting` (`RayCast`), `Breadth First Search` (`BFS`), `Comparison Sort` (`Sort`), `Dictionary`, `Minimum Spanning Forest` (`MSF`), `Remove Duplicates` (`RemDup`), `Nbody` are from the CMU Problem Based Benchmark Suite (PBBS benchmark suite) [36]. Note that the PBBS benchmark programs were originally written using Cilk Plus [19] and are converted into OpenMP. These benchmark programs cover a broad range of

real-world applications, such as computational biology, graphics, basic building blocks, finance, computer vision, and physics simulation algorithms. They also include different representative parallel structures. For instance, `Blackscholes` performs financial analysis and is parallelized by spawning and synchronizing OpenMP tasks; `Bodytrack` is a computer vision application, which is parallelized with nested parallel for loops and has ample parallelism; In contrast, `Nbody` is a scientific application and has a more complex parallel structure with both parallel for loops and spawning tasks. The different parallel structures not only affect the speedup of the benchmark programs, but also have impacts on their sensitivities to different allocated resources.

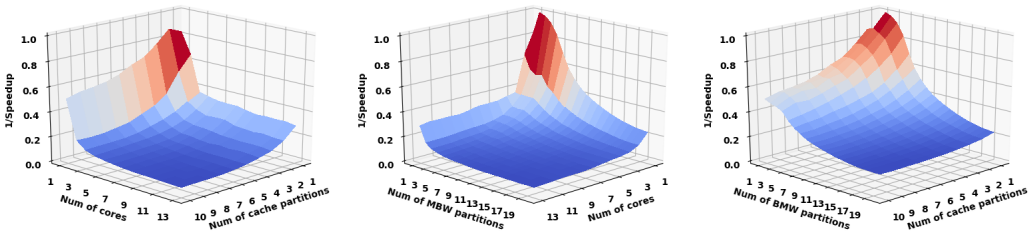
3.2 Impact of Core, Cache and Memory Bandwidth Allocations

The goal of this empirical study is to examine how the timing behavior of real-world parallel applications changes, when they are assigned with different numbers of dedicated cores, cache partitions, and memory bandwidth partitions. For brevity, we use ***MBW*** or *bandwidth* to refer to memory bandwidth in the rest of the paper.

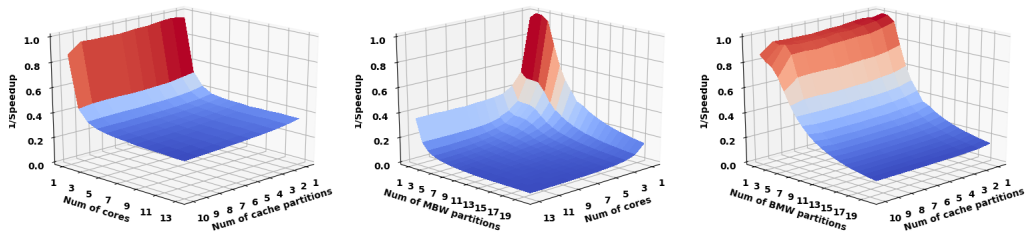
Experiment. We run each benchmark program on increasing numbers of cores, cache partitions, and memory bandwidth partitions. Under each resource allocation, we measure the execution time of the benchmark. The profiling is conducted in a setup that mimics the execution environment of co-running multiple tasks on their dedicated resources and creates as much system-level interference as possible. Specifically, we run each benchmark under profiling at a high real-time priority. Additionally, we co-run another interfering parallel task at a lower real-time priority and allocate all the remaining resources to this task. Extending the method in [45, 53], this interfering program is essentially a parallel cache-bomb and memory-intensive program that we develop by parallelizing the Stream Benchmark [31] (similar to the benchmarks in [45]) using OpenMP. This program generates intensive memory access requests by performing read and write operations on long-vectors with minimum data re-use (either in registers or in cache). In addition, it runs in parallel on the assigned cores by OpenMP parallel for loops, which frequently synchronizes using the underlying Linux futexes. In this way, it not only compete with the benchmark under profiling on the shared DRAM controller and MSHRs, but also tries to generate some contention over the internal kernel data structures related to futexes.

Ideally, one would run each benchmark program hundreds or thousands of times to measure the ***worst-case execution time*** (***WCET***) of a benchmark. However, with 13 cores, 10 cache partitions, and 20 memory bandwidth partitions, there are a total of 2,600 distinct combinations of resource allocations. Moreover, some benchmark programs take tens of seconds for one run. Hence, the measurement of one benchmark for the 2,600 combinations takes up to half a day, even when we run it once for each resource allocation. This circumstance motivates us to develop a regression analysis in Section 3.3, which enables using a much smaller number of measurements to guide the initial resource allocation for tasks. In this study, our focus is on the variety of benchmark applications and the trend of execution times upon different resource allocations, instead of obtaining the safe WCET values. Hence, we only measure the execution time of each benchmark under each of the 2,600 combinations once.

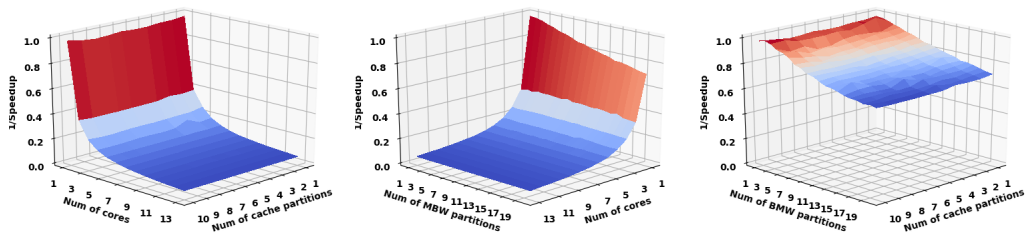
Results. Figure 1 shows the measurement results of four representative benchmarks. In particular, `RayCast` is a graphics rendering algorithm that uses the geometric algorithm of ray tracing to render and calculate the first intersecting triangles for rays that penetrate a 3-dimension bounding box containing a set of triangles. `BFS` performs a breadth-first search on a graph with a reasonably large size. `Blackscholes` calculates the prices for options using the Black-Scholes partial differential equation, which involves expensive computation on relatively small data. `Nbody` calculates the motion of particles under the influence of mutual gravitational forces in a dynamic system.



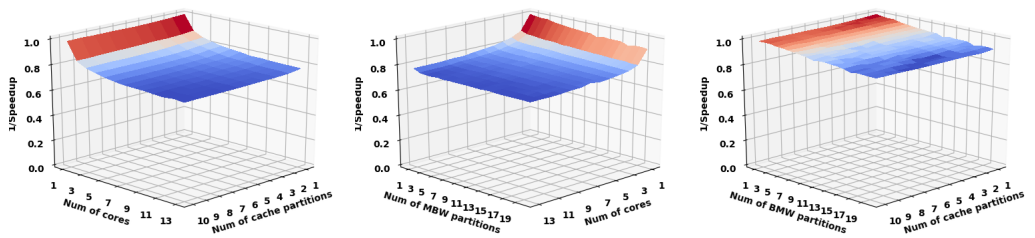
(a) RayCast: #bandwidth = 1. (b) RayCast: #cache = 1. (c) RayCast: #core = 1.



(d) BFS: #bandwidth = 1. (e) BFS: #cache = 1. (f) BFS: #core = 1.



(g) Blackscholes: #bandwidth = 1. (h) Blackscholes: #cache = 1. (i) Blackscholes: #core = 1.



(j) Nbody: #bandwidth = 1. (k) Nbody: #cache = 1. (l) Nbody: #core = 1.

Fig. 1. The slowdown (i.e., 1/speedup) of benchmark programs when assigning different numbers of cores, cache partitions, and memory bandwidth (MBW) partitions.

We calculate the speedup of a benchmark program upon a particular resource allocation. The ***speedup*** is defined as the ratio between the execution time measured for this resource allocation and the execution time when assigning only one core, one cache partition, and one memory bandwidth partition, the latter of which is also the maximum execution time of this benchmark.

For better readability, we plot Figure 1 in terms of *slowdown*, which is the inverse of speedup. The comparison between these three representative benchmarks reveals the following observation.

Observation 1. *The impact of the core, cache partition, and memory bandwidth allocations varies across different parallel benchmark applications.*

Not surprisingly, the timing behaviors of different applications vary, since they have different characteristics. For example, Figures 1(a), 1(d), and 1(g) show the speedup of the three benchmarks on increasing numbers of cores and cache partitions, while the number of memory bandwidth partitions is fixed to one. By comparing them, we can see that cache barely makes any effect on the execution times of `Blackschole` and has a slightly larger impact on `BFS`, while it significantly affects the execution times of `RayCast`. A similar trend can be seen for memory bandwidth partitions in Figures 1(b), 1(e), and 1(h). When memory bandwidth allocation increases, the running times of `RayCast` and `BFS` decrease dramatically. In contrast, memory bandwidth has little impact on `Blackschole`, especially when it is running on more than 2 cores. This is because both `RayCast` and `BFS` perform computation on large data, while the intensive computation of `Blackscholes` is performed on much smaller data.

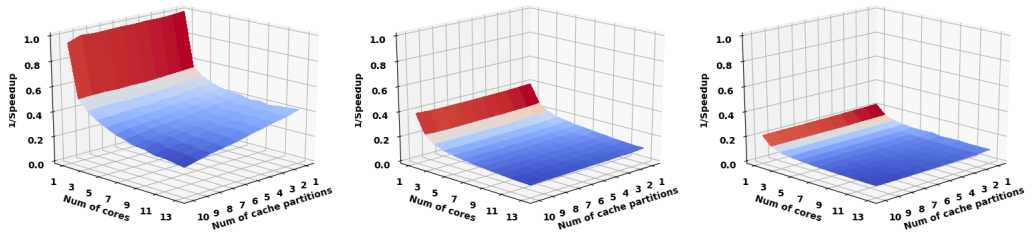
In general, well-written parallel tasks are sensitive to cores in most cases. In fact, in many cases, the execution times decrease the fastest when increasing the number of allocated cores. But the specific speedup achieved by an application when running on multiple cores depends on the parallelism of the application. For example, `Blackschole` has ample parallelism and is able to achieve near-linear speedup. In comparison, `Nbody` only has about a 25% reduction in execution times when assigned with 13 cores.

Depending on the characteristics of applications, some (e.g., `RayCast`) are sensitive to both cache and memory bandwidth resources, and some are only sensitive to memory bandwidth (e.g., `BFS`), while the others are not sensitive to cache nor memory bandwidth (e.g., `Blackschole` and `Nbody`). Somewhat surprisingly, from the benchmark applications that we profiled, we do not find any benchmark that is more sensitive to cache than to memory bandwidth. We suspect that this is both related to the memory footprint and the memory access pattern of an application.

Based on the above findings, we classify all the 12 benchmark applications into 3 large categories: cache- and MBW-sensitive benchmarks, MBW-sensitive benchmarks, and cache- and MBW-insensitive benchmarks. As shown in Table 1, out of the 12 benchmark programs, there are 2 cache- and MBW-sensitive benchmarks, 6 MBW-sensitive benchmarks, and 6 cache- and MBW-insensitive benchmarks. Note that there is a continuous spectrum from being sensitive to both cache and memory bandwidth to being sensitive to memory bandwidth only. Even for those benchmarks that are considered MBW-sensitive, increasing the number of allocation cache partitions can still reduce its execution times, albeit very slightly.

Observation 2. *The impacts of cache and memory bandwidth partitions on the execution times of a parallel benchmark are correlated.*

Not surprisingly, this observation is also similar to what was observed for sequential tasks [48]. Figure 2 presents the speedup of `Facesim` under different resource allocations. We can see that increasing the number of cache partitions reduces the execution times of `Facesim` more when given 1 memory bandwidth partition; whereas increasing the number of cache partitions reduces its execution times extremely slightly when given 10 or 20 memory bandwidth partitions. This is because when a task receives little memory bandwidth, it can be throttled frequently due to running out of bandwidth reservation. Increasing the cache size can reduce the frequency of memory accesses, and thus reduce the frequency of being throttled. In contrast, the time spent on



(a) Facesim: #bandwidth = 1. (b) Facesim: #bandwidth = 10. (c) Facesim: #bandwidth = 20.

Fig. 2. The slowdown (i.e., 1/speedup) of Facesim when assigning different numbers of cores, cache partitions, and memory bandwidth (MBW) partitions.

computing dominates the overall execution time, when the memory bandwidth is abundant. We observe similar behavior for `RayCast`, `Sort`, and `Dictionary`.

Observation 3. *For a particular cache and memory bandwidth allocation, the execution time $\widehat{e}(th_i)$ of a benchmark on th_i dedicated cores follows the formula below:*

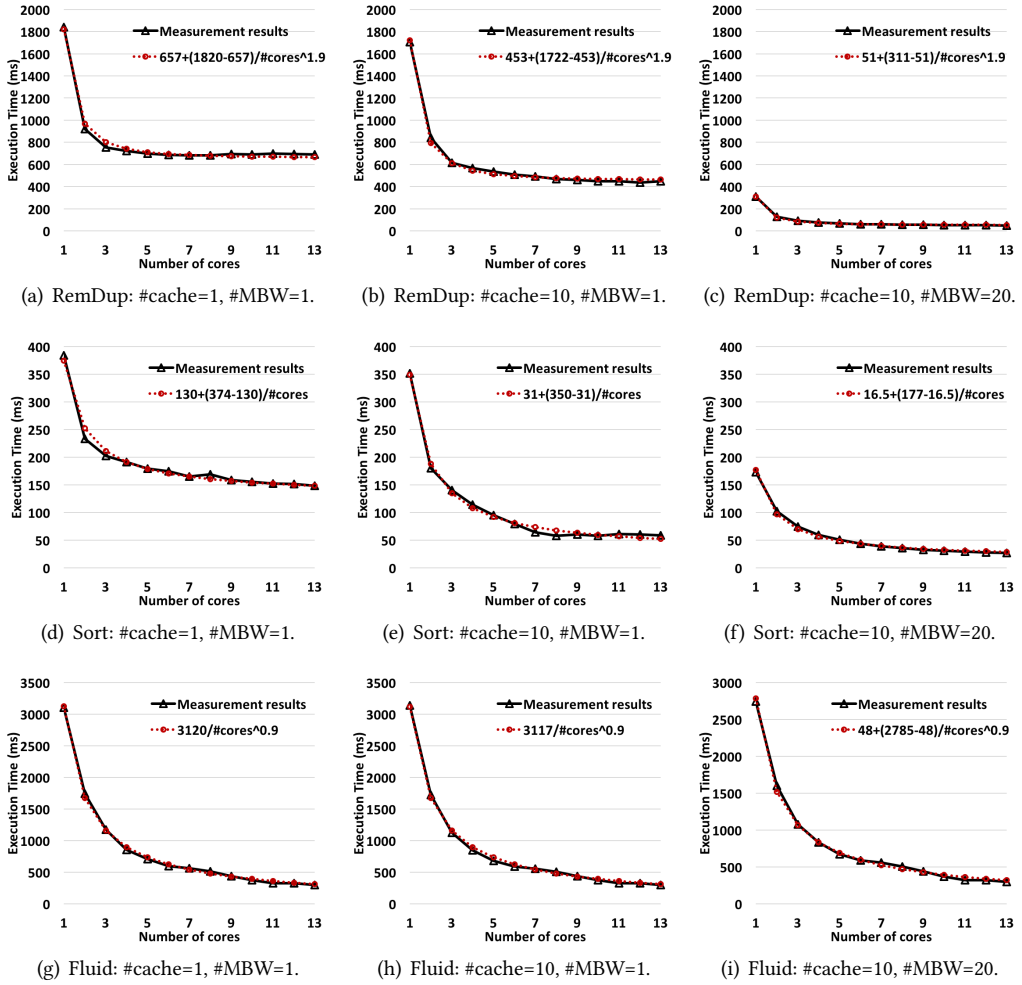
$$\widehat{e}(th_i) = f_\infty + \frac{f_1 - f_\infty}{(th_i)^c} \tag{1}$$

where c is some constant, f_1 represents the total work, and f_∞ represents the span (i.e., the execution time on an infinite number of cores) upon the particular cache and memory bandwidth allocation.

Figure 3 shows how the execution times of `RemDup`, `Sort` and `Fluid` change on increasing numbers of dedicated cores. In addition, we apply nonlinear regression using a function in the form of Equation (1) to fit the profiling results. The design of the function is inspired by the theoretical analysis of the running time $e(p)$ of a parallel program when executed by a work-conserving (i.e., greedy) scheduler on increasing numbers of cores, which is essentially following the Amdahl's Law [3]. In this analysis [28], the effects of cache and memory bandwidth are ignored, and $e_p = e_\infty + \frac{e_1 - e_\infty}{p}$, where e_1 is the total work, e_∞ is the span, and p is the number of allocated cores. Note that this classical result is almost identical to our designed function, except that the constant c in Equation (1) is always 1 under the theoretical analysis.

From Figure 3, we can first observe that `RemDup` and `Sort` are memory bandwidth sensitive benchmarks, where the execution times decrease significantly given more memory bandwidth partitions. `Fluid`, on the other hand, is insensitive to cache and memory bandwidth. Moreover, the designed function can accurately approximate the trend of the measurement results for `Sort` and the obtained constant c is 1. In contrast, to obtain a low error in the regression, the constant c is 0.9 for `Fluid` and 1.9 for `RemDup`. Unlike the classical analysis that assumes linear speedup of the parallel region, our profiling results indicate that the speedup can be superlinear or sublinear. Hence, in our nonlinear regression, we do not restrict c to 1. We also notice that allowing different values of c for the same benchmark upon different cache and memory bandwidth can slightly improve the accuracy of the regression. But the variation of c is relatively small. Hence, to reduce the number of variables in the regression, we decide to use a single constant c for the same benchmark.

Furthermore, we can see that the obtained values for f_1 and f_∞ for the same benchmark program vary a lot when different numbers of cache and memory bandwidth partitions are allocated to this benchmark. Intuitively, the cache and memory bandwidth allocation affects the latency of obtaining the data for computing, which adds to the computation time of the benchmark.



(a) RemDup: #cache=1, #MBW=1. (b) RemDup: #cache=10, #MBW=1. (c) RemDup: #cache=10, #MBW=20.

(d) Sort: #cache=1, #MBW=1. (e) Sort: #cache=10, #MBW=1. (f) Sort: #cache=10, #MBW=20.

(g) Fluid: #cache=1, #MBW=1. (h) Fluid: #cache=10, #MBW=1. (i) Fluid: #cache=10, #MBW=20.

Fig. 3. Fitting the measured execution times of RemDup, Sort, and Fluid using nonlinear regression.

3.3 Fitting WCET using Nonlinear Regression

As discussed in Section 3.2, the enormous number of resource allocation combinations causes the time to profile the worst-case execution time (WCET) of a benchmark program tremendously long, if the profiling must be done for each of the combinations. This fact motivates us to investigate applying nonlinear regression analysis to fit the measurement results of benchmarks. Our goal here is two-fold: (1) we want to see how accurate the nonlinear regression can be when a reasonable function is used; and (2) we want to see whether it is possible to perform the measurement only for a small number of combinations, apply the non-linear regression, and obtain relatively accurate estimations on the execution times.

To answer the first question, we design the following function based on Observation 3 above for estimating the (worst-case) execution times $\widehat{e}(th_i, cp_i, mp_i)$ of a benchmark program when it is

assigned with th_i cores, cp_i cache partitions, and mp_i memory bandwidth partitions:

$$\widehat{e}(th_i, cp_i, mp_i) = f_\infty(cp_i, mp_i) + \frac{f_1(cp_i, mp_i) - f_\infty(cp_i, mp_i)}{th_i^{c_{00}}} \tag{2}$$

where c_{00} is a coefficient. Similar to Equation (1), $f_1(cp_i, mp_i)$ and $f_\infty(cp_i, mp_i)$ represent the work and span of the benchmark upon cp_i cache and mp_i memory bandwidth partitions. And the coefficient c_{00} corresponds to the parameter c in Equation (1), which can be smaller, equal to, or larger than 1, as discussed under Observation 3.

The design of functions $f_1(cp_i, mp_i)$ and $f_\infty(cp_i, mp_i)$ is inspired by Observation 2. They try to capture the individual effect of cache or memory bandwidth allocations, as well as the correlation between them. Specifically, they are in the following forms:

$$\begin{aligned} f_1(cp_i, mp_i) &= c_{11} * (cp_i + c_{12})^{-c_{17}} * (mp_i + c_{13})^{-c_{18}} \\ &+ c_{14} * (cp_i + c_{12})^{-c_{17}} + c_{15} * (mp_i + c_{13})^{-c_{18}} + c_{16} \end{aligned} \tag{3}$$

$$\begin{aligned} f_\infty(cp_i, mp_i) &= c_{21} * (cp_i + c_{22})^{-c_{27}} * (mp_i + c_{23})^{-c_{28}} \\ &+ c_{24} * (cp_i + c_{22})^{-c_{27}} + c_{25} * (mp_i + c_{23})^{-c_{28}} + c_{26} \end{aligned} \tag{4}$$

where c_{11} to c_{28} are also coefficients.

Table 1. Mean Relative Error of Fitting WCET

Cache- and MBW-sensitive benchmarks:

Benchmark	RayCast	Facesim
MRE	0.05283	0.04706

MBW-sensitive benchmarks:

Benchmark	BFS	Sort	Dictionary	MSF	RemDup
MRE	0.08200	0.03881	0.09334	0.0526	0.06961

Cache- and MBW-insensitive benchmarks:

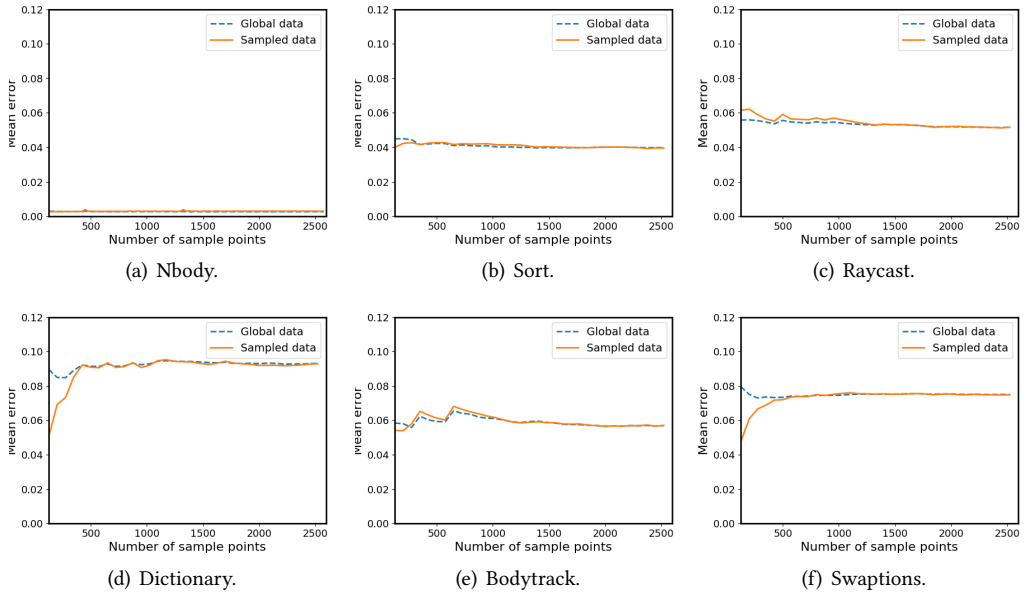
Benchmark	Bodytrack	Blackscholes	Fluidanimate	Nbody	Swaption
MRE	0.03749	0.05648	0.03948	0.00272	0.07791

Results. We use the nonlinear regression tool of `curve_fit` in Python `scipy.optimize` library to fit the measured execution times of benchmark programs. Other nonlinear regression tools, such as Matlab Cftool, DataFit from Oakdale Engineering, Origin from OriginLab, and 1stOpt from 7D-Soft, can also be used. We do not observe any difference in the results in terms of relative errors when using different tools. We initialize all the coefficients (i.e., c_{00} to c_{28}) to 1, since this initialization often leads to faster convergence in practice. One could also initialize the coefficients to any other random values. As long as the regression converges and the relative error is small, the values of the coefficients are similar. To evaluate the performance of the nonlinear regression, we calculate the relative error of the approximated execution times from the measured execution times for each allocation setting and report the ***mean relative error*** (***MRE***). Formally, mean relative error is calculated as

$$MRE = \frac{1}{n} \sum_{1}^{n} \frac{|\widetilde{e}(th_i, cp_i, mp_i) - \widehat{e}(th_i, cp_i, mp_i)|}{\widehat{e}(th_i, cp_i, mp_i)}$$

Table 1 summarizes the mean relative errors of the nonlinear regression results for different benchmark programs. Results show that the mean relative errors of the fitted execution times are smaller than 10% for all benchmarks. The accuracy of the regression does not seem to be correlated

to the type of benchmark programs. Note that although our experiments apply the nonlinear regression to the measurement of execution time for one run under each allocation setting, in principle, this approach is applicable to the WCET measurements of multiple runs.



(a) Nbody. (b) Sort. (c) Raycast.

(d) Dictionary. (e) Bodytrack. (f) Swaptions.

Fig. 4. Mean relative error of fitting the measured execution times of different benchmarks when increasing the number of sampled data points used for the nonlinear regression: the solid line is the "global" MRE calculated using all the 2,600 data points, while the dashed line is the "local" MSE calculated using only the sampled data points.

Regression using a smaller number of sampled data points. Here, we explore whether using only a small number of measurements suffices to achieve comparable accuracy with using the measurement results of all resource allocation settings. In particular, we start with feeding the regression with only 125 sampled data points. The samples come from the execution times when assigning [1, 3, 5, 7, 9] cache partitions, [1, 5, 10, 15, 20] memory bandwidth partitions, and [1, 4, 7, 10, 13] cores. The 125 initial samples are specifically chosen to evenly span the entire space. Next, we randomly sample 75 data points and add them into the regression, with the hope that more data can improve the accuracy. We repeat this process until all the 2,600 data points have been added to the regression.

Figure 4 presents the mean relative errors for 6 representative benchmark programs when increasing the number of sampled data points used for the nonlinear regression. Note that in practice, one can only calculate the mean relative error using the sampled data points. Hence, in addition to reporting the mean relative errors calculated using all the 2,600 data points (i.e., global data), we also report the mean relative errors calculated using only the sampled data points.

First and foremost, we can see that, for all benchmarks, using only about 250 data points the nonlinear regression can already achieve comparable performance to using all the data points. Therefore, our designed regression function gives the potential of significantly fewer measurements for soft real-time systems. For hard real-time systems, after obtaining the candidate resource allocation for a task set, one can conduct extensive profiling of WCET upon the allocated resources.

If the WCET ends up exceeding the deadline, then local refinement and profiling can be performed to adjust the resource allocation till all tasks can meet their deadlines.

We also observe that the trends of the MRE calculated using the global data and using the sample data are not necessarily similar. This leads to a natural question of when to stop sampling more data points (i.e., conducting more measurements). For benchmarks like `Nbody`, the initial 125 data points already achieve a very low error, which means that the regression function can very nicely approximate the true data. For the other benchmark programs, sampling another small amount of data points as a validation set and using it to determine when to stop can be a good choice.

4 PROBLEM SPECIFICATION AND PRIOR RESULTS

The empirical study of benchmarks motivates the importance of holistic resource allocation for parallel real-time tasks. This section presents the formal model for this scheduling problem based on the timing characteristics of tasks observed in our measurements.

System model. We consider a machine with N_{th} cores, sharing an L3 cache with N_{cp} equal partitions and a memory bus with N_{mp} equal memory bandwidth partitions. We extend the federated scheduling [28] introduced in Section 2 for scheduling parallel real-time tasks to incorporate cache and MBW allocation. In particular, federated scheduling forces all low-utilization tasks to run sequentially. We assume that these tasks are scheduled either by the partitioned *earliest deadline first* (*EDG*) algorithm or by the partitioned *rate monotonic* (*RM*) algorithm on its partitioned core. Each of the remaining high-utilization tasks is allocated with some dedicated cores, where it runs in parallel. The resource allocation is done via the class of service (CLOS). A CLOS can either be associated with one parallel task with a set of dedicated cores, or with one core with sequential tasks partitioned to it. Additionally, a CLOS is assigned to a set of dedicated cache partitions and a number of dedicated MBW partitions. The minimum number of cache and MBW partitions assigned to a CLOS is one.

Task model. We seek to schedule a set of m tasks. Each task τ_k is modeled as a 3-tuple $\tau_k = \{e_k(th_i, cp_i, mp_i), p_k, d_k\}$, where p_k is its period, d_k is its deadline, and $e_k(th_i, cp_i, mp_i)$ is the measured WCET of the task when executed alone on th_i cores with cp_i cache and mp_i MBW partitions. In this work, we focus on tasks with implicit deadlines where $p_k = d_k$. Similar to [48], we define $re_k = e_k(1, N_{cp}, N_{mp})$ as the ***reference WCET*** and calculate the ***reference utilization*** as $ru_k = e_k(1, N_{cp}, N_{mp})/p_i$. In addition, we also denote $pe_k = e_k(1, 1, 1)$ as the ***peak WCET*** of τ_k and calculate the ***peak utilization*** as $pu_k = e_k(1, 1, 1)/p_i$. Thus, the ***speedup*** of a task under a certain resource allocation is $t\_speedup_k(th_i, cp_i, mp_i) = pe_k/re_k$. A task is schedulable if it can always finish its execution before its deadline, and a task set is schedulable if all tasks are schedulable.

Problem/Objective. For a multicore machine with a shared L3 cache and memory bus, our goal is to develop a strategy for allocating resources, including cores, cache, and memory bandwidth partitions, to parallel real-time tasks, so that the task set is schedulable.

Most relevant work. The theoretical modeling, as well as the allocation strategy for low-utilization tasks on the partitioned cores, follows the CaM proposed by Xu et al. [48]. Here, we briefly introduce the high-level strategy of ***CaM***, which is slightly modified and used as the subroutine of our proposed algorithm in Section 6. It first uses a clustering algorithm to group the tasks by their sensitivity. It then tries to put tasks of the same group onto the same core, while maintaining the reference utilizations of cores roughly the same. Cache and MBW partitions are assigned to the over-utilized cores that have the maximum decrease in their total utilizations of the partitioned

tasks. Finally, it iteratively moves tasks from over-utilized cores to under-utilized ones and re-assigns cache and MBW partitions, until all cores become schedulable or it exceeds the maximum allowed iterations.

The main difference between the models in this work and in [48] is that tasks are parallel and may need to run on multiple cores to meet their deadlines. To handle parallel tasks, we proposed to extend federated scheduling by holistically allocating cache and MBW resources. The original ***federated scheduling*** assigns dedicated cores to parallel tasks with ***high-utilizations*** (i.e., utilization larger than one), forces the remaining ***low-utilization*** tasks to run sequentially, and partition these sequential tasks on the remaining cores.

Challenge. Incorporating cache and MBW resources for federated scheduling introduces several challenges: (1) how to distinguish high- and low-utilization tasks when a task has different utilizations given different numbers of cache and MBW partitions; (2) how to allocate a reasonable combination of cores, cache, and MBW partitions to a high-utilization task; (3) how to reserve enough cores, cache, MBW partitions for the set of low-utilization tasks.

5 OPTIMAL ALGORITHM.

In this section, we present our ***mixed-integer nonlinear programming*** (***MINLP***) with nonlinear constraints for this resource allocation problem. Constructing the MINLP and its corresponding constraints is not straightforward. One of the reasons is that high-utilization tasks are allocated with dedicated cores while low-utilization tasks are partitioned onto shared cores. For example, one needs to create MINLP variables to decide and distinguish whether a task is high- or low-utilization. Furthermore, variables are also needed to distinguish whether a core is shared by some low-utilization tasks or dedicated to a high-utilization task. The core shared by low-utilization tasks needs to be allocated with at least one cache partition and one memory bandwidth partition, in order to execute tasks. In contrast, the cores that are dedicated to a high-utilization task share the cache and memory bandwidth partitions allocated to that task.

We address all the above challenges and develop the following MINLP formulation, which can be solved using the existing solver in SCIP Optimization Suite [14]. The notations used in the MINLP formulation are summarized in Table 2. All the variables except for the last one are non-negative integers. To illustrate the intuitions for the MINLP formulation, we use the simple task set in Figure 5 as an example.

		Task 1	Task 2	Task 3	Task 4	$CPcore_j$	$MPcore_j$	Core Util
	$\zeta_{i,j}$	$\beta_1 = 0$	$\beta_2 = 1$	$\beta_3 = 0$	$\beta_4 = 0$			
Core 1	$\gamma_1 = 0$				1	1	2	0.6
Core 2	$\gamma_2 = 1$		1					
Core 3	$\gamma_3 = 0$	1		1		4	8	0.8
Core 4	$\gamma_4 = 1$		1					
$CPtask_i$			1			N_{cp}=6		
$MPtask_i$			2				N_{mp}=12	
Task Util			0.7*2					

Fig. 5. An example task set with 4 tasks on 4 cores with 6 cache partitions and 12 memory bandwidth partitions. The empty cells have a value of 0 for the corresponding variable.

At a high level, the MINLP uses the binary variable β_i to distinguish whether a task is high-utilization (needs dedicated resources) or low-utilization (partitioned to a core shared with other low-utilization tasks). For instance, in Figure 5 Task 2 is considered as a high-utilization task. Note that a task may change from high to utilization by increasing the allocated cache and memory partitions to reduce its execution time. Similarly, γ_j specifies whether a core is dedicated to a high-util task or shared by low-util tasks. For example, Core 3 is shared by Tasks 1 and 3. The mapping between the tasks and cores is stored in $\zeta_{i,j}$, which is highlighted in green in Figure 5. Depending on whether a core is shared by tasks and whether a task is assigned with multiple cores, we can distinguish the type of a core and the type of a task, respectively.

The critical reason for the need for separate indicator variables to distinguish the type of tasks and cores is that the cache and memory bandwidths associated with them are different. In particular, low-utilization tasks that are partitioned onto the same core share the cache and bandwidth partitions associated with this core (e.g., Tasks 1 and 3 on Core 3 share 4 cache and 8 bandwidth partitions). In contrast, a high-utilization task has dedicated cache and bandwidth partitions (e.g., Task 2 on Cores 2 and 4 has 1 cache and 2 bandwidth partitions), which are essentially shared by this task's dedicated cores. Hence, the constraints for cache and bandwidth partitions can only be properly formulated when the task and core types are clear. Such information is specified by $CPtask_i$ and $MPtask_i$ for high-utilization tasks, and by $CPcore_j$ and $MPcore_j$ for the cores shared by low-utilization tasks. All these requirements are encoded into the MINLP constraints to obtain the optimal solution to the resource allocation problem.

Table 2. Notations

m	Number of tasks in the task set
N_{th}	Number of cores on the hardware
N_{cp}	Number of L3 cache partitions on the hardware
N_{mp}	Number of memory bandwidth partitions on the hardware
α_i	Binary: if task i has a valid resource allocation
β_i	Binary: if task i is high-util with dedicated resources; or, low-util task
γ_j	Binary: if core j is dedicated to a high-util task; or, shared by low-util tasks
$\zeta_{i,j}$	Binary: if task i executes on core j
$CPtask_i$	Number of cache partitions allocated to (high-util) task i
$MPtask_i$	Number of bandwidth partitions allocated to (high-util) task i
$CPcore_j$	Number of cache partitions allocated to (low-util) core j
$MPcore_j$	Number of bandwidth partitions allocated to (low-util) core j
$e_i(TH_i, CP_i, MP_i)$	Measured WCET of task i when executed on the specified resources

We now formally describe the constraints that our MINLP formulation encodes.

C1) This constraint essentially distinguishes whether a task has a valid resource allocation using properties of the task-core mapping $\zeta_{i,j}$. The variable α_i is only used in the optimization objective to maximize the number of schedulable tasks. When task i does not have a valid resource allocation (i.e., $\alpha_i = 0$), then no cores should have been allocated to this task. When there exists a valid allocation, the number of assigned cores should not exceed the total available cores N_{th}.

$$\forall 1 \leq i \leq m: \quad \alpha_i \leq \sum_{j=1}^{N_{th}} \zeta_{i,j} \leq N_{th} \cdot \alpha_i$$

C2) In the task-core mapping $\zeta_{i,j}$, high-utilization tasks (i.e., $\beta_i = 1$) cannot share the same core j. Similarly, a low-utilization task i (i.e., $1 - \beta_i = 1$) cannot be assigned to more than one core.

$$\forall 1 \le j \le N_{th}: \quad \sum_{i=1}^{m} (\zeta_{i,j} \cdot \beta_i) \le 1$$

$$\forall 1 \le i \le m: \quad \sum_{j=1}^{N_{th}} \left(\zeta_{i,j} \cdot (1 - \beta_i)\right) \le 1$$

C3) This constraint essentially distinguishes the type of a core using properties of the task-core mapping. In the task-core mapping $\zeta_{i,j}$, if core j is dedicated (i.e., $\gamma_j = 1$), there is exactly one high-utilization task (i.e., $\beta_i = 1$) executing on this core. Otherwise, this core is shared, so no high-utilization task should execute on this core.

$$\forall 1 \le j \le N_{th}: \quad \sum_{i=1}^{m} (\zeta_{i,j} \cdot \beta_i) = \gamma_j$$

C4) This constraint essentially distinguishes the type of a task using properties of the task-core mapping. If task i executes on one or multiple dedicated cores (i.e., $\gamma_j = 1$), this task must be high-utilization (i.e., $\beta_i = 1$).

$$\forall 1 \le i \le m: \quad \sum_{j=1}^{N_{th}} \left(\zeta_{i,j} \cdot \gamma_j\right) \le \beta_i \cdot N_{th}$$

C5-1) The next four sets of constraints restrict the allocation of cache and bandwidth partitions. First, a dedicated core (i.e., $\gamma_j = 1$) must share the partitions with the cores belonging to the same high-utilization task, so it does not have any exclusive partitions.

$$\forall 1 \le j \le N_{th}: \quad \gamma_j \cdot CPcore_j = 0 \quad \wedge \quad \gamma_j \cdot MPcore_j = 0$$

C5-2) Second, a shared core (i.e., $\gamma_j = 0$) must have at least one cache partition and one bandwidth partition.

$$\forall 1 \le j \le N_{th}: \quad \gamma_j + CPcore_j \ge 1 \quad \wedge \quad \gamma_j + CPcore_j \ge 1$$

C5-3) Next, a low-utilization task (i.e., $\beta_i = 0$) does not have any allocated cache or bandwidth partition. For high-utilization tasks, the number of allocated partitions is bounded by availability.

$$\forall 1 \le i \le m: \quad CPtask_i \le \beta_i \cdot N_{cp} \quad \wedge \quad MPtask_i \le \beta_i \cdot N_{mp}$$

C5-4) Lastly, a high-utilization task (i.e., $\beta_i = 1$) must be allocated with at least one cache partition and one bandwidth partition.

$$\forall 1 \le i \le m: \quad CPtask_i \ge \beta_i \quad \wedge \quad MPtask_i \ge \beta_i$$

C6) The next two constraints bound the total number of allocated cache (or bandwidth) partitions.

$$\sum_{i=1}^{m} (\beta_i \cdot CPtask_i) + \sum_{j=1}^{N_{th}} \left(\gamma_i \cdot CPcore_j\right) \le N_{cp}$$

$$\sum_{i=1}^{m} (\beta_i \cdot MPtask_i) + \sum_{j=1}^{N_{th}} \left(\gamma_i \cdot MPcore_j\right) \le N_{mp}$$

C7-1) The last sets of constraints make sure the task is schedulable. For a high-utilization task i that is allocated with $\sum_{j=1}^{N_{th}} \zeta_{i,j}$ cores, $CPtask_i$ cache, and $MPtask_i$ bandwidth partitions, its execution time e_i should be no more than its implicit deadline p_i. In addition, we also require that this task is allocated with the minimum number of cores, i.e., reducing one dedicated core would result in deadline misses.

$$\forall 1 \leq i \leq m: \quad \beta_i \cdot e_i \left(\sum_{j=1}^{N_{th}} \zeta_{i,j}, CPtask_i, MPtask_i \right) \leq p_i$$

$$\forall 1 \leq i \leq m: \quad \beta_i \cdot e_i \left((\sum_{j=1}^{N_{th}} \zeta_{i,j} - 1), CPtask_i, MPtask_i \right) \geq p_i$$

C7-2) For a core with $CPcore_i$ allocated cache and $MPcore_i$ bandwidth partitions shared by low-utilization tasks (i.e., $1 - \gamma_j = 1$), the total utilization of these tasks cannot exceed 1.

$$\forall 1 \leq j \leq N_{th}: \quad (1 - \gamma_j) \cdot \sum_{i=1}^{m} \left((1 - \beta_i) \cdot \zeta_{i,j} \cdot \frac{e_i(1, CPcore_j, MPcore_j)}{p_i} \right) \leq 1$$

Objective: Our goal is to maximize the number of tasks that can be feasibly scheduled. This can be formulated as follows using the indicator variable α_i:

$$maximize \sum_{i=1}^{m} \alpha_i$$

Improved MINLP implementation. While being optimal, the complexity of solving this MINLP problem is extremely high, making it very inefficient to use in practice. Moreover, the existing solvers for MINLP need to convert the MINLP to regular nonlinear programming before adding the integer constraints, so the non-continuous and nonlinear $e_k(th_i, cp_i, mp_i)$ cannot be directly used. Instead, we must use Function (2) obtained from fitting the measured data, which further reduces the efficiency. Thus, we further improve the implementation of the MINLP formulation.

In particular, we separate the allocation to high- and low-utilization tasks and use a brute-force method to enumerate all the good allocations for high-utilization tasks. Once the allocation decisions for the high-utilization tasks are made, the remaining problem becomes the resource allocation problem for sequential (low-utilization) tasks. For this problem, there exists a mixed-integer programming (MIP) formulation [48], which is much faster than the original MINLP formulation for the entire task set.

Thus, the key to a good improved implementation is on deciding (1) which tasks are high-utilization tasks and (2) how many resources should be allocated to high-utilization tasks. To maintain the optimality of the MINLP formulation, all possible choices for the above two questions must be verified before determining that the task set is unschedulable. However, not all choices have equal importance — the ordering of these choices crucially affects the running time of the implementation for most task sets. This is because once a schedulable allocation decision is found, all the remaining choices no longer need to be verified.

With this intuition, we first construct all possible subsets of tasks in the original task set. If one subset is chosen as high-utilization tasks, the supplement subset becomes low-utilization tasks. Among these possible subsets, we sort them according to their total reference utilization from large to small. Thus, the tasks with higher utilizations will be considered as high-utilization tasks first. Next, for a considered subset, we verify all combinations of resource allocation to these high-utilization tasks and see if any combination can make all tasks schedulable. Here, we

prune some combinations that are clearly not feasible or reasonable. For example, if a task is not schedulable given a set of resources, then it is also not schedulable given strictly fewer resources. Moreover, among all the combinations that are schedulable, it is obviously more beneficial to the remaining low-utilization tasks if strictly more resources remain. Hence, instead of running the MIP formulation of the low-utilization tasks for all schedulable combinations, we prune those that use strictly more resources. In this way, only the combinations at the Pareto boundary are verified, which significantly reduces the running time of this implementation in practice.

6 HOLISTIC RESOURCE ALLOCATION FOR FEDERATED SCHEDULING

Although the improved MINLP implementation reduces the running time by a lot, it can still take a long time to obtain the optimal results, especially when the problem size is large. Therefore, we propose a heuristic-based strategy that has comparable or slightly worse performance with a running time in orders of magnitude shorter than the MINLP formulation. This section first gives an overview of our holistic resource allocation strategy and then provides details of the algorithm.

6.1 Algorithm Overview

The holistic resource allocation algorithm for parallel real-time tasks leverages the advantages of CaM [48] and federated scheduling [28], as well as insights obtained from our empirical study using real-world benchmarks in Section 3. In particular, our holistic algorithm is based on the following high-level strategies.

(1) We assign dedicated cores, cache partitions, and memory bandwidth partitions to each high-utilization task;
(2) We use the Speedup profile $t\_speedup_k(th_i, cp_i, mp_i)$ to cluster low-utilization tasks to try to partition tasks with similar sensitivity onto the same core, while also maintaining relatively balanced workload between low-utilization cores;
(3) We use a dynamical threshold for distinguishing high-utilization and low-utilization tasks;
(4) We assign each available resource instance to the unschedulable task or core that receives the largest positive benefit (i.e., the largest reduction in its utilization);
(5) We allow an unschedulable core with low-utilization tasks to exchange resources with a schedulable high-utilization task, under the condition that the high-utilization task remains schedulable and the low-utilization core has a reduction in its total utilization after the exchange.

Algorithm 1 gives the main steps of our holistic resource allocation algorithm, which composes of five phases:

(1) Phase 1 (Lines 1–3) dynamically classifies all tasks into high- and low-utilization tasks, by iteratively moving more tasks from low-utilization to high-utilization set. The set of high-utilization tasks τ^{high} is initialized as the tasks with high reference utilizations $ru_k = e_k(1, N_{cp}, N_{mp})/p_i \geq 1$. At each iteration, one more low-utilization task with the largest reference utilization is selected by the *selectAddHT()* procedure to be moved from τ^{low} to τ^{high}.

(2) Phase 2 (Lines 4–7) assigns cores, cache and MBW partitions to each high-utilization task τ_k in τ^{high} until it can meet its deadline (i.e., $e_k(th_i, cp_i, mp_i) \leq d_k$). The allocation follows the largest benefit first, to be explained in Section 6.2. During the assignment, any task in τ^{low} that was considered as low-utilization tasks but can no longer meet its deadline by running sequentially on the remaining resources for low-utilization tasks is moved to τ^{high} and participate the resource assignment process. If the available resources are not even enough for high-utilization tasks, the system is deemed unschedulable. Otherwise, the procedure *initLR()* calculates the unallocated resources to be used for partitioning low-utilization tasks.

(3) Phase 3 (Lines 8–13) tries to cluster the low-utilization tasks based on their speedup profile, partition them on the remaining cores, and assign cache and MBW partitions to each core using the procedure *CaMAllocLR*. It is almost the same as the heuristic resource allocation algorithm in CaM [48], except for some differences in its balance procedure. In this procedure, a low-utilization task can be migrated from one core to another. Here, we need to make sure that after the migration, this task still remains as a low-utilization task and its original core does not become empty. If all low-utilization tasks are schedulable on their partitioned cores, the task set is schedulable.

(4) Phase 4 (Lines 14–15) is only reached if there is some core with unschedulable low-utilization tasks. In this case, via procedure *resEx()* the unschedulable low-utilization cores try to use one or two types of resources to exchange for the other resource from high-utilization tasks that could bring more benefit to these cores.

(5) Phase 5 (Lines 16–18) checks the schedulability of the low-utilization tasks. If the low-utilization cores still cannot schedulable all the low-utilization tasks after the exchange, it is possible that they contain some tasks with relatively high utilization. Thus, the next iteration considers moving one more task to the set of high-utilization tasks. Otherwise, the task set is schedulable.

Algorithm 1: Holistic Resource Allocator

Input: τ: task set; N_{th}: total number of cores; N_{cp}: total number of cache partitions; N_{mp}: total number of MBW partitions; *maxKM*: the maximum iterations for KMeans; *maxPerm*: the maximum iterations.
Output: Schedulable or Unschedulable.

1 $\{\tau^{high}, \tau^{low}\} \leftarrow initHighTask(\tau, N_{cp}, N_{mp})$
2 **for** $m_{addH} = 0$ **up to** $size(\tau) - size(\tau^{high})$ **by** $+1$ **do**
3 $\{\tau^{high}, \tau^{low}\} \leftarrow selectAddHT(\tau^{high}, \tau^{low}, m_{addH})$
4 $\{\tau^{high}\} \leftarrow allocHR(\tau^{high}, N_{th}, N_{cp}, N_{mp})$
5 $sched = checkSched(\tau^{high})$
6 **if** $sched = unschedulable$ **then break**;
7 $\{N_{th}^{low}, N_{cp}^{low}, N_{mp}^{low}\} \leftarrow initLR(\tau^{high}, N_{th}, N_{cp}, N_{mp})$
8 sort($clusters \leftarrow clusterTasks(\tau^{low}, N_{th}, maxKM)$)
9 $sched = unschedulable$
10 **for** $j = maxPerm$ **down to** 0 **by** -1 **do**
11 $perm\_clusters \leftarrow permute(clusters)$
12 $\{cores^{Sched}, cores^{USched}, sched\} \leftarrow CaMAllocLR(perm\_clusters, N_{th}^{low}, N_{cp}^{low}, N_{mp}^{low})$
13 **if** $sched = schedulable$ **then break**;
14 $\tau^{high}, cores^{USched} \leftarrow resEx(\tau^{high}, cores^{USched})$
15 $cores^{low} = cores^{Sched} \cup cores^{USched}$
16 $sched = checkSched(cores^{low})$
17 **if** $sched = schedulable$ **then break**;
18 **if** $sched = schedulable$ **then break**;

6.2 Procedures of the Algorithm

We now give details to the two main procedures in Algorithm 1.

Procedure *allocHR()* allocates resources to each high-utilization task for meeting its deadline. The minimum numbers of cores th_{min}, cache partitions cp_{min}, and MBW partitions mp_{min} for task τ_k are calculated such that $e_k(th_{min}, N_{cp}, N_{mp}) > d_k$, $e_k(N_{th}, cp_{min}, N_{mp}) > d_k$, and $e_k(N_{th}, N_{cp}, mp_{min}) > d_k$. After initializing the minimum resources, the procedure calculates the maximum benefit of allocating one instance of resource to one unschedulable task. For example, the benefit of allocation an additional core to τ_k with assignment (th_i, cp_i, mp_i) is calculated as $u_k(th_i + 1, cp_i, mp_i) - u_k(th_i, cp_i, mp_i)$. The benefit of allocating one cache or MBW partition can be calculated similarly. Among all the possible allocations with different available resources and different unschedulable tasks, the procedure will choose the one that results in the largest benefit (i.e., the largest reduction in utilization). Hence, this choice of allocation best utilizes the resources for high-utilization tasks.

Procedure *resEx()* tries to exchange resources between unschedulable cores partitioned with low-utilization tasks and schedulable high-utilization tasks. Note that although Procedure *allocHR()* makes good allocation decisions for high-utilization tasks, such a decision may not be globally optimal for low-utilization tasks. For example, the cache may only result in a slightly better benefit than MBW for high-utilization tasks, but *allocHR()* will assign all the available cache to high-utilization tasks. With no cache left, the low-utilization tasks cannot be scheduled on the remaining cores. At a high level, Procedure *resEx()* enumerates all valid resource exchange plans between a low-utilization core and a high-utilization task. An exchange will only happen if the high-utilization task remains schedulable and the low-utilization core has utilization benefits after the exchange.

Specifically, there are three steps in this procedure. First, if there are some empty cores not used by any low-utilization tasks, then each of the unschedulable low-utilization cores tries to use these empty cores to exchange for the cache and memory bandwidth resources from each of the high-utilization tasks. This exchange is successful and will take place if both of them are schedulable after the exchange. Note that some cores can be empty while some other low-utilization cores are unschedulable. This is mainly because there does not remain at least one cache partition and one memory bandwidth partition to be associated with the empty core. Hence, when this happens, the empty cores can be used by high-utilization tasks that are already allocated with some cache and memory bandwidth. With extra cores, these high-utilization tasks may return some cache and memory bandwidth partitions, while being schedulable. These resources can then be used by the unschedulable low-utilization cores to improve their schedulability.

Second, if there is no empty core available, unschedulable low-utilization cores can try to use some of their cache and memory bandwidth resources to exchange one or more cores from high-utilization tasks. Lastly, we can also exchange the cache and memory bandwidth resources between low-utilization and high-utilization tasks. This may help balance these two types of resources and obtain the minimum number of cache partitions and memory bandwidth partitions needed for scheduling the low-utilization tasks on their cores. As a result, each core can be assigned the minimum resources.

6.3 Complexity of the Algorithm

We first discuss the complexity of the subprocedures and then summarize the total complexity of the algorithm. The following procedures iterate over all tasks and take $O(m)$ time: *initHighTask()*, *selectAddHT()*, *checkSched()*, and *initLR()*. The *allocHR()* procedure enumerates all unschedulable high-utilization tasks for finding the one with the maximum benefit, and the number of times that this process repeats is at most the number of remaining resources, so this procedure takes

$O(m \cdot (N_{th} + N_{cp} + N_{mp}))$ time. The *sort()* procedure takes $O(m \log m)$ time. Since the *maxKM* and *maxPerm* are predetermined constants, the *clusterTask()* procedure iterates over all clusters for all tasks for a constant number of iterations (i.e., $O(m \cdot N_{th})$). The *permute()* procedure takes a constant time. The complexity of the *CaMAllocLR()* procedure depends on the number of low-utilization tasks and the remaining resources. In the worst-case, all tasks are low-utilization, which takes $O(\max\{m \log m, m \cdot N_{th}, N_{th} \cdot N_{cp}^2 \cdot N_{mp}^2\})$ time. The *resEx()* procedure takes $O(m \cdot \max\{N_{th}, N_{cp}, N_{mp}\}^2)$ in the worst-case. Therefore, the entire algorithm has a complexity of $O(m \cdot \max\{m \cdot (N_{th} + N_{cp} + N_{mp}), m^2 \log m \cdot N_{th}, N_{th} \cdot N_{cp}^2 \cdot N_{mp}^2, m \cdot \max\{N_{th}, N_{cp}, N_{mp}\}^2\})$.

7 NUMERICAL EVALUATION

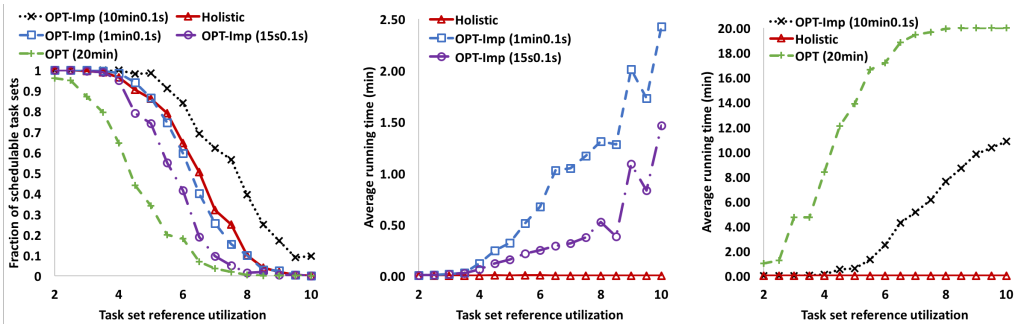
In this section, we conducted numerical experiments to evaluate our proposed holistic resource allocation algorithms on task sets randomly generated using the profiling results of realistic parallel benchmark programs in Section 3.

7.1 Experimental Setup

Workload generation. To evaluate the scalability of our proposed algorithms, we consider four types of hardware systems. The smallest one has 13 cores with 10 cache partitions and 20 MBW partitions, which is consistent with our real hardware platform described in Section 3. A slightly larger one has twice the amount of resources, i.e., 26 cores with 20 cache partitions and 40 MBW partitions. We also consider the one with three times the resources (i.e., 39 cores with 30 cache partitions and 60 MBW partitions) and the largest one with four times the resources (i.e., 52 cores with 40 cache partitions and 80 MBW partitions).

We vary the total reference utilization of task sets from 2 to the number of available cores m. For each total utilization, we randomly generate 200 task sets with the desired utilization. We generate a task's reference utilization uniformly at random from the utilization range from 0.2 to $0.5\sqrt{m}$, where m is the number of available cores. For each task, its WCET profile is randomly chosen from the empirical measurements of the 12 real-world benchmark programs described in Section 3. Each benchmark has an equal probability of being chosen unless otherwise specified. Due to the long profiling time discussed in Section 3.3, we were only able to measure each benchmark's execution times upon each resource allocation combination for one time, instead of multiple times for taking the worst-case value. Thus, the measured execution times have some variances. Hence, we refine the WCET profiles by making sure that the WCET upon a specific resource allocation is the same or larger than all the WCETs upon resource allocations that have strictly more allocated resources. Additionally, the measurements are performed on our real hardware platform with a limited number of resources (i.e., 13 cores with 10 cache partitions and 20 MBW partitions). In order to conduct larger-scale numerical experiments considering larger machines with more resources and more tasks, we set the WCETs upon more resources the same as the one upon 13 cores with 10 cache partitions and 20 MBW partitions.

The period (and implicit deadline) of a task is obtained by using its reference WCET to divide its reference utilization. For most of the experiment settings except for one, before adding a task to the task set, we also check whether this task can meet its implicit deadline when all the available resources of the hardware platform have been allocated to this task. If it is still not schedulable, then we do not add this task to the task set; otherwise, the task set will also become unschedulable no matter how resources are allocated to tasks.



(a) Schedulability comparison. (b) Average computation time. (c) Average computation time.

Fig. 6. Comparison with the optimal variations for task sets on 13 cores with 10 cache and 20 MBW partitions.

7.2 Schedulability Performance

To the best of our knowledge, there are no existing solutions that address the core, cache, and memory bandwidth resource allocation problem for parallel real-time tasks. Therefore, in the first experiment, we evaluate our proposed holistic algorithm by comparing it against the optimal solution that is based on solving the mixed-integer nonlinear programming problem.

In this experiment, we consider four versions of the optimal. The first is the original MINLP formulation (denoted as OPT) that is directly solved by the solver in the SCIP Optimization Suite [14]. The next is the improved implementation (denoted as OPT-Imp). As discussed in Section 5, the implementation of the optimal is extremely inefficient. Even with our improved optimal variation, the running time is still very long. Moreover, when there is no valid allocation, the solver used by all the optimal solutions will not return any result. Therefore, we need to set a timeout for the optimal solutions. For each task set, it is deemed unschedulable if the optimal algorithm does not return any result by 15s, 1min, 10min, and 20min. Figure 6(a) shows the comparison results. The original implementation with 20min running time performs the worst, simply because directly solving the general MINLP is extremely hard. To verify that our implementation is correct, for task sets that are schedulable using our holistic algorithm, we initialize the corresponding variables in the MINLP formulation as the same as our generated allocation decision. Once doing that, the MINLP solver can return in a shorter time and verify that the encoded allocation decision passes all the optimization constraints. Compared with the improved implementation, we can see that our algorithm has comparable performance with the improved optimal algorithm with 1min running time limit and outperforms those that have shorter running time limits.

7.3 Running Time Efficiency

Although our holistic algorithm achieves slightly lower schedulability than the best performing optimal variation OPT (10min0.1s), it is significantly more efficient than any of the optimal variations. As shown in Figures 6(b) and 6(c), the average running times of all the optimal variations grow at an exponential rate. For example, the OPT variation that solves a single MINLP problem needs 4min on average to obtain solutions for task sets with reference utilizations that are as low as 3. For many schedulable task sets with reference utilizations that are larger than 4.5, OPT cannot return the valid resource allocation even if running for 2 hours.

The improved optimal variations have much faster running times, but they are still many orders of magnitude slower than the holistic algorithm. For instance, when the task set reference utilization

is at least 5.5, OPT (10min0.1s) needs more than 1min on average to find the schedulable resource allocations. Its running time rapidly increases to 10min when the total reference utilization is 9.5. The fastest optimal variation OPT (10s0.1s) that has worse schedulability than the holistic algorithm takes about 1.5min to solve for task sets with reference utilization 10. In contrast, the average running times of our holistic algorithm are all below 24s. This computation efficiency gap between the optimal variations and the holistic algorithm increases when there are more available resources.

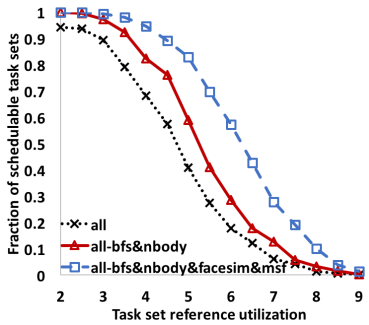


Fig. 7. Experiments with a subset of benchmarks on 13 cores.

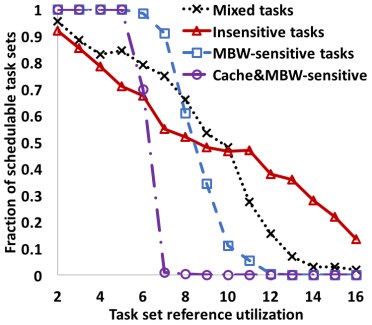


Fig. 8. Experiments with specific type of benchmarks on 26 cores.

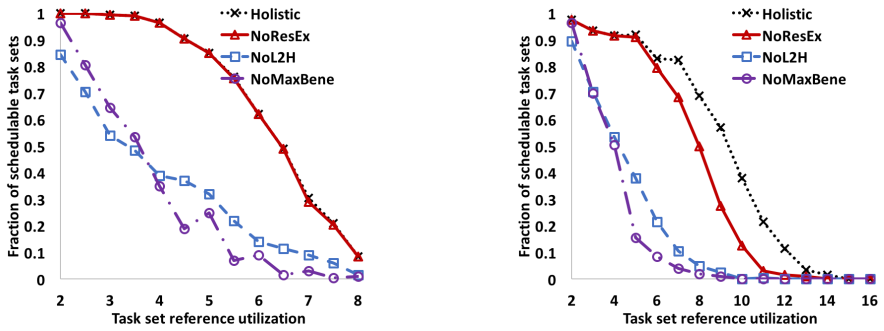
7.4 Impact of Different Benchmarks

As discussed in workload generation, in all of the other experiments, we only add a randomly generated benchmark task into the task set if this task can at least meet its deadline when monopolizing all the resources. Figure 7 motivates why it is necessary to do so. In particular, `BFS`, `Nbody`, `facesim`, and `MSF` are memory-intensive benchmarks. Because the reference utilization is defined as the WCET of a task on a single core with all the available cache and MBW partitions, these memory-sensitive tasks already have high speedup given all the MBW partitions. When their randomly generated utilizations are high, allocating all the cores to them may not achieve sufficient additional speedup to allow them to meet their deadlines. In Figure 7, we do not check the schedulability of a task upon task set generation. Instead, we selectively exclude some of these 4 benchmarks in the task set generation. The differences in schedulability, thus, reflect the impact of these benchmarks.

To study the effect of different types of benchmarks, we also conduct an experiment where we generate task sets with only one type of benchmark and compare the schedulability results to the task sets with mixed types. Figure 8 shows that the fraction of schedulable task sets drastically drops to 0 for task sets with cache- and MBW-sensitive tasks. In contrast, when there are more cache- and MBW-insensitive tasks, there is a fraction of task sets schedulable at high total reference utilizations. The noticeable difference here also implicitly and partially verifies that our classification of benchmark programs indeed distinguishes the different characteristics of tasks.

7.5 Ablation Study of Our Algorithm

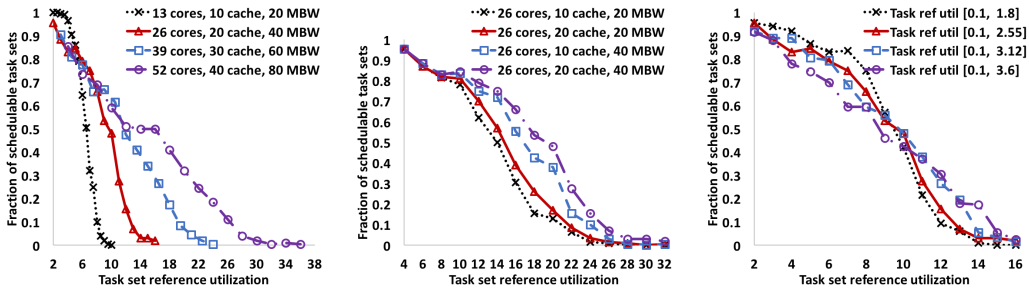
The impact of the different phases of our holistic algorithm can be observed in Figure 9(b). Specifically, version NoResEx disables the procedure *resEx()*, so there is not resource exchange between low- and high-utilization tasks. Version NoL2H disable procedure `selectAddHT()` and use a fixed threshold for classifying low- and high-utilization tasks. In contrast, NoMaxBene does not use the maximum benefit strategy in procedure `allocHR()` for finding the best resource to allocate to



(a) Fewer cache- and MBW-sensitive tasks. (b) More cache- and MBW-sensitive tasks.

Fig. 9. Ablation study of holistic algorithm on task sets with different percentages of cache- and MBW-sensitive tasks on 26 cores, 20 cache partitions, and 40 MBW partitions.

a high-utilization task and uses round-robin instead. We can see that procedure `allocHR()` and `selectAddHT()` both have large impacts on the schedulability of task sets, while procedure `resEx()` almost no impact for the task sets that select benchmarks uniformly at random in Figure 9(a). Based on our classification for benchmark programs, there are only 2 cache- and MBW-sensitive benchmarks out of the 12 benchmarks. When a task set has few or no such benchmarks, resource exchanges are unlikely to happen. To verify our hypothesis, we construct task sets that select a cache- and MBW-sensitive benchmark with a probability twice the probability of selecting the other ones. As shown in Figure 9(b), with more cache- and MBW-sensitive benchmarks, resource exchange plays a more important role in finding a good resource allocation.



(a) Increase the number of all resources. (b) Change the number of some resources. (c) Change the task utilization range.

Fig. 10. Fraction of schedulable task sets with different task set generation parameters.

7.6 Impact of Platform Configurations and Task Parameters

Finally, we also vary the task parameters and platform configurations to evaluate their impact on the schedulability of task sets. As expected, increasing the number of available resources allows task sets with higher total reference utilizations to be schedulable. And the increase in schedulability is roughly proportional to the increase in the total resources, as revealed in Figure 10(a). On the other hand, different types of resources have different levels of impact on schedulability. As shown in

Figure 10(b), decreasing the number of cache partitions by half has a smaller impact, compared to decreasing the number of memory bandwidth partitions by half. However, this is related to the characteristics of the randomly generated task sets. For example, if all tasks are insensitive to cache and memory bandwidth, the decrease in both resources will have little impact on schedulability. In Figure 10(c), we vary the range of tasks' reference utilizations. We can see that this change has a relatively small impact on the schedulability of task sets.

8 EMPIRICAL EVALUATION

To demonstrate the practicality of our proposed framework on real hardware platforms and evaluate its empirical performance, we extend the federated scheduling system [29] to support cache and memory bandwidth partitioning. The modification to the federated scheduling is similar to the modification to benchmark programs as described in Section 3.

We first measure the system overhead of different resource allocation operations, including the core allocation operation via `pthread_setaffinity_np()`, the cache partition allocation operation via CAT, and the memory bandwidth allocation via Memguard. Results presented in Table 3 show that the overhead of allocation cache partitions is significantly higher than partitioning cores and setting memory bandwidth reservations. Fortunately, our proposed framework follows the federated scheduling paradigm where the resource partitionings are performed only once before the execution of the task sets. Therefore, the high overhead only occurs once even before the execution of all tasks, does not impact the schedulability and efficiency of the task sets, and needs not to be incorporated into the analysis.

Table 3. Overhead measurement

Operation	Average overhead (ms)	Maximum overhead (ms)
Core allocation	0.142	0.152
Cache partition allocation	569.1	570.8
Memory bandwidth allocation	4.4	4.5

We choose 3 representative benchmark programs (`RayCast`, `Swaptions`, and `RemDup`), one in each type, and measure their execution times of 50 runs on all 2,600 combinations of resource allocations. Since the measured maximum execution times of 50 runs may not be the true and safe WCET, we inflate the maximum value by 1.1 and use the inflated WCET for task set generation. We follow a similar procedure to that in Section 7 to randomly generated 100 task sets for each reference utilization on 13 cores. We run each task set for a duration that is equal to 50 times the longest period of the tasks in the set.

Although when generating task sets, we inflate the measured maximum execution time by 1.1. In the experiments, we would like to evaluate how the WCET estimates used by the holistic algorithm affect the schedulability of task sets. Hence, we use three types of WCET estimates: (1) the measured maximum execution time inflated by 1.1 — namely WCETx1.1; (2) the measured maximum execution time without any inflation — namely WCETx1.0; and (3) the regression results using 250 sampled maximum execution times inflated by 1.1 — WCETx1.1+fitting.

Note that both the greedy algorithm and the MINLP algorithms cannot generate a resource allocation decision for a task set, if the task set is deemed unschedulable by the respective algorithm. Additionally, it is not clear how to modify the optimal MINLP formulations to output any allocation decision for unschedulable task sets, as the optimization constraints are violated for these task sets. Therefore, one can only run the theoretically schedulable task sets to see if there is any deadline miss observed during the actual execution on the hardware platform. These results are presented in

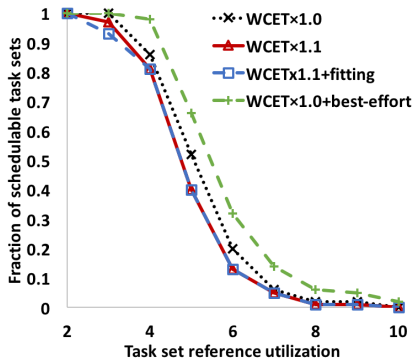


Fig. 11. Empirical experiments on 13 cores.

Figure 11 with legends WCETx1.0, WCETx1.1, and WCETx1.1+fitting. *In contrast to the numerical experiments in the previous section, here a task set is considered schedulable only if no deadline miss is observed during the actual execution of this task set on the hardware platform.*

We would also like to see how pessimistic the measured WCETs are when the workload is consolidated onto the platform following our proposed heuristic. To achieve this, we modified our holistic resource allocation strategy to generate one "reasonable" allocation decision, even when the task set is deemed unschedulable. In this way, we can execute the theoretically unschedulable task sets on the real platform to reveal the pessimism of the measured WCETs. Specifically, for every potential allocation decision tested during the process of the heuristic-based algorithm, we calculate the maximum of each core's utilization given this allocation decision and the task set. We record and output the decision that has the lowest maximum core utilization. Intuitively, this metric helps output the allocation decision that balances the load of each core. The observed schedulability of this best-effort approach (on top of the allocation decisions for the theoretically schedulable task sets) with WCETx1.0 estimates is denoted as WCETx1.0+best-effort in Figure 11.

We can observe that inflating the maximum execution times introduces a small amount of pessimism. A small number of task sets become unschedulable, because increasing the WCET estimate causes the need for more resources to meet deadlines. Moreover, comparing the results between WCETx1.0 and WCETx1.0+best-effort, we can see that a few theoretically unschedulable task sets do not encounter any deadline miss during the actual execution.

Among the experimented task sets, we do not observe any deadline miss for both WCETx1.1 and WCETx1.0. In contrast, if we directly executing the resource allocation decisions made by the algorithm using WCETx1.1+fitting estimates, there will be two tasks missing their deadlines. This is because the estimation made by the regression algorithm over-optimistically predicts the WCETs for some resource allocations. When the holistic algorithm happens to choose these allocations, the involved tasks will miss deadlines. However, as discussed in Section 3, when regression with a small number of sampled data points is used for estimating the WCETs, profiling for the chosen resource allocation must be performed. If the profiling indicates that a task cannot meet its deadline given this resource allocation, a local refinement can be made by adding more resources to this task until the profiling shows that it can meet its deadline. For the particular two task sets in our experiments, we adopt this procedure and locally refine the allocation. The two task sets become schedulable. After this procedure, we do not observe any deadline miss for WCETx1.1+fitting.

Note that the regression results can also be pessimistic. For example, there are a few task sets under reference utilization 3 that are deemed unschedulable using the WCETx1.1+fitting estimates.

In principle, a local refinement can also be applied here to greedily search for potential resource allocation and validate it using profiling. Overall, the empirical experiments verify the efficiency and effectiveness of our proposed framework for parallel real-time applications.

9 CONCLUSION

In this work, we present a holistic resource allocation strategy for parallel real-time tasks executing on multicore systems that share cache and memory bandwidth resources. Our strategy integrates existing cache partitioning and memory bandwidth regulation mechanisms and leverages results from resource allocation for sequential tasks and federated scheduling for parallel tasks. Based on the insights obtained from empirical evaluations of real-world parallel benchmarks, we develop an approach for parallel real-time tasks to improve the practicality of measurement-based models. The numerical evaluation and proof of concept implementation demonstrate that our proposed framework is efficient and practical.

As future work, we plan to extend our insights and mechanisms to global scheduling algorithms and constrained deadline tasks. In addition to partitioning and isolating memory bandwidth resources, incorporating the memory partitioning techniques into our framework is reserved for future work. Finally, we would like to explore whether dynamically allocating cache and memory bandwidth resources can further improve the performance of parallel real-time systems.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation (USA) under Grant Numbers CNS–1948457 and the National Natural Science Foundation of China under Grant No. U20A6003.

REFERENCES

[1] Ankit Agrawal, Gerhard Fohler, Johannes Freitag, Jan Nowotsch, Sascha Uhrig, and Michael Paulitsch. 2017. Contention-aware dynamic memory bandwidth isolation with predictability in COTS multicores: An avionics case study. In *Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[2] Ahmed Alhammad and Rodolfo Pellizzoni. 2016. Trading cores for memory bandwidth in real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–11.

[3] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[4] Björn Andersson and Dionisio de Niz. 2012. Analyzing Global-EDF for multiprocessor scheduling of parallel tasks. In *International Conference On Principles Of Distributed Systems*. Springer, 16–30.

[5] ARM. 2018. Memory System Resource Partitioning and Monitoring (MPAM). https://developer.arm.com/documentation/ddi0598/latest/.

[6] Muhammad Ali Awan, Konstantinos Bletsas, Pedro F Souto, Benny Akesson, and Eduardo Tovar. 2017. Mixed-Criticality Scheduling with Dynamic Redistribution of Shared Cache. In *Euromicro Conference on Real-Time Systems (ECRTS)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[7] Sanjoy Baruah. 2015. Federated Scheduling of Sporadic DAG Task Systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 179–186.

[8] Sanjoy Baruah. 2016. The federated scheduling of systems of mixed-criticality sporadic DAG tasks. In *IEEE Real-Time Systems Symposium (RTSS)*. 227–236.

[9] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University. http://parsec.cs.princeton.edu.

[10] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. 2013. Feasibility analysis in the sporadic DAG task model. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 225–233.

[11] Gang Chen, Biao Hu, Kai Huang, Alois Knoll, Di Liu, and Todor Stefanov. 2014. Automatic cache partitioning and time-triggered scheduling for real-time MPSoCs. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.

[12] Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, Arvind Easwaran, and Insik Shin. 2013. Global EDF schedulability analysis for synchronous parallel tasks on multicore platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*. 25–34.

[13] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. 2016. Ginseng: Market-driven LLC allocation. In *USENIX Annual Technical Conference (ATC)*. 295–308.

[14] Gerald Gamrath, Daniel Anderson, Ksenia Bestuzheva, Wei-Kun Chen, Leon Eifler, Maxime Gasse, Patrick Gemander, Ambros Gleixner, Leona Gottwald, Katrin Halbig, et al. 2020. The SCIP Optimization Suite 7.0. In *Technical Report*.

[15] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. 2015. A survey on cache management mechanisms for real-time embedded systems. *Computing Surveys (CSUR)* 48, 2 (2015), 1–36.

[16] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *International conference on Embedded software (EMSOFT)*. ACM, 245–254.

[17] Danlu Guo, Mohamed Hassan, Rodolfo Pellizzoni, and Hiren Patel. 2018. A comparative study of predictable DRAM controllers. *Transactions on Embedded Computing Systems (TECS)* 17, 2 (2018), 1–23.

[18] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. 2015. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 307–316.

[19] Intel. 2013. Intel CilkPlus v1.2. https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[20] Intel. 2019. User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat.

[21] Xu Jiang, Nan Guan, Xiang Long, and Wang Yi. 2017. Semi-federated scheduling of parallel real-time tasks on multiprocessors. In *Real-Time Systems Symposium (RTSS)*. IEEE, 80–91.

[22] Xu Jiang, Xiang Long, Nan Guan, and Han Wan. 2016. On the decomposition-based Global EDF scheduling of parallel real-time tasks. In *Real-Time Systems Symposium (RTSS)*. IEEE, 237–246.

[23] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. 2014. Bounding memory interference delay in COTS-based multi-core systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 145–154.

[24] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *International Conference on Embedded Software (EMSOFT)*. IEEE, 1–10.

[25] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Raj Rajkumar. 2013. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *4th International Conference on Cyber-Physical Systems (ICCPS)*. 31–40.

[26] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. 2010. Scheduling parallel real-time tasks on multi-core processors. In *31st IEEE Real-Time Systems Symposium (RTSS)*. 259–268.

[27] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2013. Analysis of Global EDF for Parallel Tasks. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*. 3–13.

[28] J. Li, Jian-Jia Chen, K. Agrawal, C.Lu, C.D. Gill, and Abusayeed Saifullah. 2014. Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks. In *26th Euromicro Conference on Real-Time Systems (ECRTS)*. 85–96.

[29] Jing Li, Son Dinh, Kevin Kieselbach, Kunal Agrawal, Christopher Gill, and Chenyang Lu. 2016. Randomized Work Stealing for Large Scale Soft Real-time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*. 203–214.

[30] Yonghui Li, Benny Akesson, and Kees Goossens. 2016. Architecture and analysis of a dynamically-scheduled real-time memory controller. *Real-Time Systems* 52, 5 (2016), 675–729.

[31] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *Computer society technical committee on computer architecture (TCCA) newsletter* 2, 19-25 (1995).

[32] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. 2016. Schedulability analysis of conditional parallel task graphs in multicore systems. *IEEE Trans. Comput.* 66, 2 (2016), 339–353.

[33] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. 2012. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*. 321–330.

[34] Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut. 2019. Cache-conscious off-line real-time scheduling for multi-core platforms: algorithms and implementation. *Real-Time Systems* 55, 4 (2019), 810–849.

[35] OpenMP. 2013. OpenMP Application Program Interface v4.0. http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[36] PBBS. 2014. Problem Based Benchmark Suite. http://www.cs.cmu.edu/~pbbs.

[37] Rodolfo Pellizzoni and Heechul Yun. 2016. Memory servers for multicore systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–12.

[38] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. 2013. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems* 49, 4 (2013), 404–435.

[39] Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. 2015. Static task partitioning for locked caches in multicore real-time systems. *Transactions on Embedded Computing Systems (TECS)* 14, 1 (2015), 1–30.

[40] Mahadev Satyanarayanan. 2017. The emergence of edge computing. *Computer* 50, 1 (2017), 30–39.
[41] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *Internet of Things Journal* 3, 5 (2016), 637–646.
[42] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. 2020. E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management. In *Real-Time Systems Symposium (RTSS)*. IEEE, 345–357.
[43] Corey Tessler, Venkata P Modekurthy, Nathan Fisher, and Abusayeed Saifullah. 2020. Bringing inter-thread cache benefits to federated scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 281–295.
[44] Niklas Ueter, Georg von der Bruggen, Jian-Jia Chen, Jing Li, and Kunal Agrawal. 2018. Reservation-Based Federated Scheduling for Parallel Real-Time Tasks. In *IEEE Real-Time Systems Symposium (RTSS)*. 482–494.
[45] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. 2016. Taming non-blocking caches to improve isolation in multicore real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 1–12.
[46] Qi Wang and Gabriel Parmer. 2014. FJOS: Practical, Predictable, and Efficient System Support for Fork/Join Parallelism. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE 20th*. 25–36.
[47] Jun Xiao, Sebastian Altmeyer, and Andy Pimentel. 2017. Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches. In *Real-Time Systems Symposium (RTSS)*. IEEE, 199–208.
[48] Meng Xu, Linh Thi Xuan Phan, Hyon-Young Choi, Yuhan Lin, Haoran Li, Chenyang Lu, and Insup Lee. 2019. Holistic resource allocation for multicore real-time systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 345–356.
[49] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. 2016. Maracas: A real-time multicore vCPU scheduling framework. In *Real-Time Systems Symposium (RTSS)*. IEEE, 179–190.
[50] Heechul Yun, Waqar Ali, Santosh Gondi, and Siddhartha Biswas. 2016. Bwlock: A dynamic memory access control framework for soft real-time applications on multicore platforms. *Transactions on Computers (TC)* 66, 7 (2016), 1247–1252.
[51] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. 2014. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 155–166.
[52] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2013. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 55–64.
[53] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. 2015. Memory bandwidth management for efficient performance isolation in multi-core platforms. *Transactions on Computers (TC)* 65, 2 (2015), 562–576.
[54] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *European conference on Computer systems*. ACM, 89–102.
[55] Yanqi Zhou and David Wentzlaff. 2016. MITTS: Memory inter-arrival time traffic shaping. *SIGARCH Computer Architecture News* 44, 3 (2016), 532–544.
[56] Alexander Zuepke and Robert Kaiser. 2019. Deterministic futexes: Addressing WCET and bounded interference concerns. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 65–76.