Addressing Multi-Core Timing Interference using Co-Runner Locking

Hyoseung Kim*, Dionisio de Niz†, Bjorn Andersson†, Mark Klein†, John Lehoczky‡
*Electrical and Computer Engineering, University of California, Riverside
†Software Engineering Institute, Carnegie Mellon University
‡Department of Statistics and Data Science, Carnegie Mellon University
hyoseung@ucr.edu,{dionisio,baandersson,mk}@sei.cmu.edu, jl16@andrew.cmu.edu

***Abstract*—This paper presents a task synchronization mechanism, called co-runner locking, to address the timing interference problem in multi-core real-time systems. It prevents certain subsets of tasks from executing simultaneously on different cores in order to avoid large performance penalties from inter-core interference. We provide the general properties of the co-runner locking mechanism and discuss the runtime control policies that determine the execution order of tasks in a co-runner-locking relationship. For schedulability analysis, we derive a response-time test that upper-bounds the delay from co-runner locking and the slowdown imposed by permitted co-runners by combining two new analytic approaches: job-oriented and load-oriented. In evaluation, we demonstrate that the co-runner locking mechanism is an effective alternative to address the "one-out-of-m" problem and brings about a significant improvement in real-time taskset schedulability.**

I. Introduction

Timing interference in multi-core processors has been a major barrier to the efficient use of the latest commodity parallel platforms in safety-critical domains. Due to hardware resources shared among processing cores, such as caches and memory buses, the execution time of a task on one core can be adversely affected by other *co-runner* tasks on different cores. The degree of slowdown caused by such interference is subject to the type of tasks and the combination of co-runner task sets, but the worst-case impact could be enormous as reported in the literature. For example, in a quad-core platform, the slowdown can be as large as $12\times$ due to contention on a shared cache [29] or DRAM memory [25], and more than $300\times$ due to writebuffer blocking [9]. This leads to the so-called "one-out-of-m" problem [31] that leaves only one core's capacity usable for real-time task execution in a m-core system. The significance of this problem has been recognized by certification authorities and industry vendors, e.g., CAST-32A [46] for avionics systems.

Many prior studies from the real-time systems community have presented techniques to tackle the timing interference delay in modern multi-core platforms. In particular, resource partitioning and reservation for caches [28, 56, 58], DRAM banks [25, 54, 62], and memory bandwidth [59, 64] have been considered effective ways to reduce the amount of delay and enhance the level of timing predictability. Efforts have been also made to analytically capture this delay in schedulability analysis, by assuming the worst-case contention on shared hardware resources [17, 25, 63]. However, these approaches are inherently pessimistic as partitioned resources could be underutilized and the assumed "worst case" might not actually occur in a given system. Time-triggered non-preemptive scheduling has been studied to statically determine the occurrence of possible co-runners [48, 50, 51], but it is not applicable to priority-driven preemptive scheduling which is the de-facto standard of today's real-time operating systems.

In this paper, we propose *co-runner locking*, a task synchronization-based approach to address the multi-core timing interference problem. The key idea of co-runner locking is simple: it prevents subsets of tasks from running simultaneously so that unwanted worst-case scenarios (e.g., extremely high slowdown due to memory-intensive tasks) can be avoided at runtime. In other words, a task can declare a co-runner lock to enforce a mutually-exclusive relationship with selected co-runner tasks in making scheduling decisions.[1] This approach is particularly useful given that the extreme slowdowns reported in the literature are caused by carefully-engineered synthetic tasks [9, 25, 29, 31] and thus only a few tasks are likely to cause the one-out-of-m problem in real-world scenarios. All other scheduling decisions follow the conventional preemptive priority-based scheduling. Hence, tasks are preemptible at any time by priority, and the progress of tasks that are not specified by co-runner locking is unaffected. We define the properties of the co-runner locking scheme and discuss the runtime execution control policies. Although our focus is on partitioned fixed-priority scheduling in this paper, the general idea of co-runner locking is applicable to dynamic-priority scheduling.

For schedulability analysis, our system model is based on the co-runner dependent execution time model [6], where the timing interference imposed by a given set of co-runners is characterized by a slowdown factor for the worst-case execution time obtained in isolation. We derive a response-time test that jointly considers two new analytical approaches, job-oriented and load-oriented, to get a tighter bound than the baseline approach that simply multiplies the worst-case execution time by the maximum slowdown. We then present algorithms to determine which tasks should use co-runner

[1] This is the difference from conventional locks that check mutually-exclusive conditions at the boundaries of critical sections in the code, whereas co-runner locking does so at scheduling points and has no predefined critical section.

locking. Experiments based on random tasksets and a real system implementation show that the proposed co-runner locking scheme benefits schedulability and mitigates performance penalties induced by contention on shared resources, thereby serving as an effective alternative to existing methods.

II. Related Work

From the advent of multi-core architectures, many techniques have been developed to safely account for and minimize timing interference in real-time system design. A comprehensive survey on various techniques is provided in [37]. For shared caches, the most popular approach is cache partitioning [8, 26, 28, 58]. With this, a fraction of the shared cache is either allocated exclusively to one task or permitted to be used by a subset of tasks. Cache locking has been studied along with partitioning [38, 56]. Real-time cache management has also been extended to the virtualization environment [23, 29, 36, 57, 60]. For memory-induced interference, DRAM bank partitioning [25, 62] and memory bandwidth regulation [59, 64] are proposed. These help reduce the worst-case memory interference delay, thereby improving temporal isolation and predictability. Researchers have incorporated such resource partitioning techniques into schedulability analysis to upper-bound memory interference delay [17, 25, 63], and resource allocation algorithms to improve schedulability [24, 42, 59]. Partitioning both caches and DRAM banks has also been studied [32, 54]. It is worth noting that we are *not* trying to replace these resource partitioning techniques or claiming that our work is superior to them. Instead, we propose an alternative, scheduling-based approach that can be used when the above methods are unavailable (e.g., cache/DRAM partitioning requires significant modifications to the virtual memory subsystem and suffers from memory size and fragmentation issues [26, 54]), or together with these methods. Our work can help existing cache/DRAM partitioning and bandwidth regulation techniques since they can reduce the degree of interference but cannot eliminate it (e.g., $> 5\times$ slowdown remains even after cache and DRAM partitioning [25]). In other works, after resource partitioning and contention-aware task allocation, co-runner locking can be applied to the tasks that still have a large slowdown.

Researchers have also proposed scheduling approaches to address the multi-core interference problem. The first type of work is *contention-free* scheduling [5, 10, 15, 21, 44, 53, 61], which strictly prevents simultaneous execution of tasks if they cause any contention on shared resources. PREM [44, 53, 61] decomposes each task into memory and computation phases, and serializes memory phases of all tasks. Calandrino and Anderson [15] proposed a cache-aware scheduler for soft real-time systems, which heuristically makes scheduling decisions at every time quantum so that the working set size of jobs does not exceed the cache size. RT-Gang [5] schedules one parallel real-time task as a gang at a time so that the interference penalty of each gang can be contained within the gang and does not propagate to other gangs. The follow-up work [4] introduced *virtual* gangs which group synchronously-released tasks with the same period into the same gang. Similarly, Isolation Scheduling (IS) [21] groups tasks into time partitions and schedules each partition exclusively on a multi-core platform.

The second type of work is *interference-aware* scheduling, which aims to bound or minimize timing penalties by using the notion of interference-sensitive WCET (*is*WCET) [40, 47, 48, 50, 51]. The *is*WCET of a task is determined by its possible co-runners for a given schedule (i.e., the duration of overlapped execution with such co-runners), thereby reducing pessimism in estimating the interference penalty. Nowotsch et al. [40] proposed an analysis method to compute *is*WCET considering the worst-case overlap scenario, and implemented it using AbsInt aiT [3]. Rouxel et al. [48] proposed non-preemptive time-triggered scheduling that minimizes the makespan of tasks with *is*WCET. Skalistis and Kritikakou [50, 51] extended this idea for runtime adaptation, which executes tasks earlier than the original schedule when there is a slack. Such adaptation works since *is*WCET already takes into account the worst-case relative phasing of co-runners. Andersson et al. [6] proposed the co-runner dependent execution time model that generalizes *is*WCET using a progress speed parameter (inverse of slowdown) and developed schedulability analysis for sporadic tasks under preemptive fixed-priority partitioned scheduling. Progress/slowdown parameters can be obtained by the measurement-based method presented in that paper, *is*WCET analysis [40, 48], or analytical interference bounds [24, 63]. Recent work like [52] can also be used to identify time-varying shared resource demands from measurements. Our work is inspired by these approaches and thus falls into the second type of work. Unlike the work for non-preemptive time-triggered scheduling [48, 50, 51], we focus on priority-based preemptive scheduling. Compared with [6], our work proposes co-runner locking that offers a way to prevent excessive slowdown scenarios. Co-runner locking may be seen as similar to the gang approach [4, 21] in the sense that both give a control over co-runner execution; however, our work does not necessitate co-runners to have the same period and the same release offset.

Real-time task synchronization has been primarily studied for critical sections protected by mutex locks. Many protocols have been developed for multi-core and multi-processor systems [12, 14, 19, 30, 39, 45, 55], and applied to shared GPUs hardware accelerators [18, 20, 27, 43, 49]. The proposed co-running locking also synchronizes task execution for given mutually-exclusive conditions, but at scheduling points rather than critical section boundaries. We will discuss how co-runner locking could be used with conventional synchronization protocols for tasks with critical sections in Sec. IV-D.

Recently, a scheduler feature called *core scheduling* has been introduced in Linux [35], Xen [2], VMWare ESXi [1], and MS Hyper-V [7], as a countermeasure to address Spectre vulnerability attacks [33] on simultaneous multithreading (SMT). Core scheduling allows the user to specify what tasks can share a physical core with SMT. While this can be thought of as a form of co-runner locking, core scheduling itself does not provide a formal execution model, control policy, or real-

time schedulability analysis.

III. System Model

We consider a shared-memory multi-core system where tasks are scheduled by partitioned fixed-priority scheduling. Hence, each task is statically assigned to one CPU core and does not migrate to another core at runtime. We use M to denote the number of cores in the system and Γ_p to denote the taskset of a core p, i.e., $\Gamma = \bigcup_{1 \leq p \leq M} \Gamma_p$. The system has shared hardware resources, and the contention on these resources causes a slowdown to task execution. CPU cores may be heterogeneous in that they may operate at different clock speed or a subset of cores is grouped into a cluster with different shared resources.[2] In this case, task execution time and the degree of slowdown may vary depending on the allocation of tasks to CPU cores, but we assume that task allocation is given and fixed by the system designer.

The entire taskset Γ of the system comprises n sporadic real-time tasks with constrained deadlines, i.e., $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. Each task has a unique priority, which can be assigned by any fixed-priority assignment policy, e.g., Rate Monotonic (RM) or Deadline Monotonic (DM), with an arbitrary tie-breaking rule. Task τ_i is characterized as follows:

$$\tau_i := (C_i, T_i, D_i)$$

- C_i: The worst-case execution requirement of any job of τ_i. If τ_i executes with no slowdown (e.g., running in complete isolation), its worst-case execution time is equal to C_i.
- T_i: The minimum inter-arrival time between any two jobs
- D_i: The relative deadline ($D_i \leq T_i$)

While we do not consider tasks with critical sections, the proposed co-runner locking approach can be used with conventional multiprocessor real-time synchronization protocols like MPCP [45]. This will be discussed in Sec. IV-D.

Our task model follows the co-runner dependent execution time model [6]. Hence, the worst-case execution time (WCET) of a task can be greater than C_i and determined by a slowdown from co-runner tasks. Below we define key parameters to represent co-runner tasks and their slowdown factors.

Def. 1. *S is the set of non-empty sets of tasks that can potentially execute in parallel in the system. In other words, S includes non-empty combinations of tasks from each per-core taskset, formally given by*

$$S = \{s = s_1 \cup s_2 \cup ... \cup s_N \mid s \neq \emptyset \wedge s_p \subset \Gamma_p \wedge |s_p| \leq 1\}$$

For example, consider τ_1, τ_2, and τ_3 in a dual-core system. Assume that τ_1 and τ_2 are assigned to core 1 and τ_3 is assigned to core 2. Then, S of this system is as follows: $S = \{\{\tau_1\}, \{\tau_2\}, \{\tau_3\}, \{\tau_1, \tau_3\}, \{\tau_2, \tau_3\}\}$. Each element of S represents a set of tasks that may run simultaneously in the system at a specific point in time.

[2]The hardware platform we use for evaluation, Nvidia AGX Xavier, belongs to this category since it has four CPU clusters, each with two cores sharing one L2 cache (i.e., total eight cores and four L2 caches), and one L3 cache and main memory shared among all eight cores.

Def. 2 (Potential co-runner sets)**.** *S_i is the set of potential co-runner sets of a task τ_i. Formally, it is defined as:*

$$S_i = \{s = s' \setminus \{\tau_i\} \mid s' \in S \wedge \tau_i \in s'\}$$

For the above example, $S_1 = \{\emptyset, \{\tau_3\}\}$, $S_2 = \{\emptyset, \{\tau_3\}\}$, $S_3 = \{\emptyset, \{\tau_1\}, \{\tau_2\}\}$.

Def. 3 (Slowdown)**.** *$\sigma_{i,s} \geq 1$ is the worst-case slowdown factor for a task τ_i due to a co-runner set s (if $s = \emptyset$, $\sigma_{i,s} = 1$). Hence, the worst-case execution time of τ_i in the presence of the co-runner set s is $C_i \cdot \sigma_{i,s}$.*

We use σ_i to denote the set of all slowdown factors for a task τ_i, i.e., $\sigma_i = \{\sigma_{i,s} \mid s \in S_i\}$. These can be obtained by the methods discussed in Sec. II. As an illustration, the above example taskset is assumed to have the following slowdown factors: $\sigma_1 = \{\sigma_{1,\emptyset} = 1, \sigma_{1,\{\tau_3\}} = 2\}$, $\sigma_2 = \{\sigma_{2,\emptyset} = 1, \sigma_{2,\{\tau_3\}} = 1.5\}$, $\sigma_3 = \{\sigma_{3,\emptyset} = 1, \sigma_{3,\{\tau_1\}} = 3, \sigma_{3,\{\tau_2\}} = 2.5\}$.

It is worth noting that, in practice, the system designer does not need to obtain the slowdown factors of all possible co-runner sets for each task. One can give a default slowdown value to unknown/untested co-runner sets, e.g., by an automatic tool to find the maximum slowdown [22], architecture-level bounds [41, 48], or the degree of isolation offered by cache and DRAM partitioning [25, 63] when they are used with co-runner locking. One can even set $\sigma = \infty$ for certain co-runners if they should never execute in parallel at any time. Of course, a higher number of precise slowdown estimates can lead to better schedulability, but the flexibility of the model facilitates its applicability to real systems.

IV. Co-Runner Locking Scheme

This section defines the properties of the proposed co-running locking scheme and discusses the execution control policies that can be used for tasks with co-runner locking.

A. Definition and Properties

At the heart of the co-runner locking scheme is representing and enforcing the mutually-exclusive conditions for selected co-runner tasks. We thus introduce a co-runner exclusion set for a task τ_i, denoted by ϵ_i, as follows.

Def. 4 (Co-runner exclusion set)**.** *ϵ_i is the set of co-runner tasks that are **not** allowed to execute in parallel with τ_i. The relationship is symmetric but not transitive, e.g., $\tau_j \in \epsilon_i \implies \tau_i \in \epsilon_j$ (τ_i and τ_j are prohibited to run simultaneously), and $\tau_j \in \epsilon_i \wedge \tau_i \in \epsilon_k \not\implies \tau_j \in \epsilon_k$. If there is no restriction on co-runner tasks of τ_i, then $\epsilon_i = \emptyset$ and $\forall \tau_j : \tau_i \notin \epsilon_j$.*

With the co-runner exclusion set ϵ_i, not all potential co-runner sets $s \in S_i$ will execute in parallel with τ_i at runtime. Hence, we use G_i to define a set of "true" co-runners of τ_i.

Def. 5 (True co-runner sets)**.** *G_i is the set of true co-runner sets of τ_i which are not prevented by ϵ_i and thus can execute in parallel at runtime. It is formally given by:*

$$G_i = \{s \mid s \in S_i \wedge \nexists \tau_j (\tau_j \in s \wedge \tau_j \in \epsilon_i)\}$$

Hence, if a co-runner task τ_j is in ϵ_i, the task τ_i does not experience slowdown from a co-runner set s' that includes τ_j. In other words, a slowdown factor of $\sigma_{i,s'}$ does not apply to any job of τ_i if $s' \cap \epsilon_i \neq \emptyset$. Only $\sigma_{i,s}$ with $s \cap \epsilon_i = \emptyset$ applies.

Def. 6 (Slowdown with co-runner locking). *ρ_i is the set of slowdown factors for the true co-runner sets in G_i.*

$$\rho_i = \{\sigma_{i,s} \mid s \in G_i\}$$

Def. 7 (Maximum slowdown). *θ_i is the maximum slowdown factor in ρ_i, i.e., $\theta_i = \max_{\sigma_{i,s} \in \rho_i} \sigma_{i,s}$.*

With the above definitions, one can imagine that the worst-case execution time of τ_i can be easily bounded by $C_i \cdot \theta_i$ for any occurrence of unprevented co-runners. In Sec. V, we will take this approach to derive a baseline analysis and then propose a more precise method.

B. Execution Control Policy

The co-runner exclusion set ϵ_i does not determine the execution order of co-runners when they compete. Hence, we need a runtime policy to control their execution. Below defines a priority-based execution control policy for tasks with co-runner locking.

R1. When a ready task τ_i is chosen for execution on a core p, the scheduler checks whether there is any task $\tau_k \in \epsilon_i$ that is currently running on a different core q.

a) If there is no τ_k, then τ_i begins execution on the core p.
b) If there is τ_k and it has lower priority than τ_i, then τ_k is suspended immediately until τ_i's completion, and after that, the scheduler executes τ_i on the core p and the next ready task ($\neq \tau_k$) on the core q.
c) If there is τ_k and it has higher priority than τ_i, then the task τ_i is suspended and the scheduler finds the next ready task for execution on the core p.

R2. When a task τ_i that is currently executing on a core p stops execution (due to job completion, suspension, or preemption), the following conditions are considered.

a) If there are any tasks in ϵ_i that have been suspended due to τ_i's execution, the scheduler wakes up all such tasks (so they become ready to run). Later, when they are selected for execution by the scheduler, R1 will take place.
b) If no task in ϵ_i has been suspended due to τ_i, the scheduler executes the higher-priority ready task on the core p (same as regular priority-based scheduling).

In R2.a, the reason the scheduler wakes up all suspended tasks instead of just one highest-priority task is that two or more suspended tasks may be eligible to execute simultaneously with other co-runners when they get CPU cores. This is different from the semantics of conventional critical sections, which have a fixed access count for each resource.

Example. Consider a taskset $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ on three CPU cores, as shown in Fig. 1. Tasks are ordered in decreasing order of priorities, so τ_1 on core 1 has the highest priority and τ_4 on core 2 has the lowest priority. Each task has a non-empty co-runner exclusion set ϵ, which has been determined to prevent

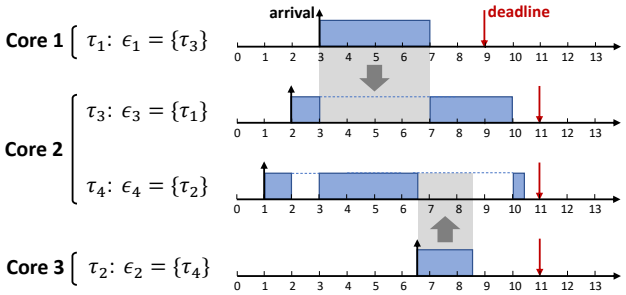


Fig. 1: Co-runner locking with priority-based execution control

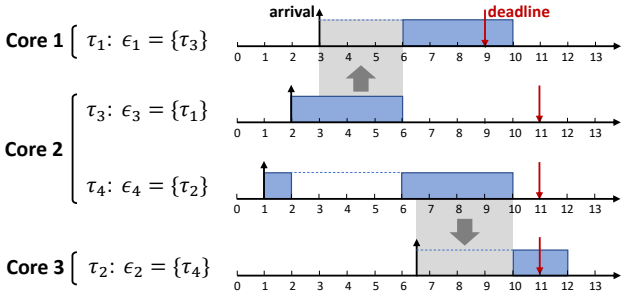


Fig. 2: Co-runner locking with FCFS execution control

slowdown from particular co-runner tasks. Any other tasks not in ϵ are assumed to incur zero slowdown for simplicity. For instance, τ_1 has $\tau_3 \in \epsilon_1$ and τ_1's execution time is unaffected by the parallel execution of τ_2 and τ_4.

Let us take a look at the scheduling timeline of the figure. When τ_1 arrives at time 3, the execution control policy finds out that $\tau_3 \in \epsilon_1$ is running and has lower priority than τ_1. Hence, it suspends τ_3 and starts τ_1's execution, as if τ_1 preempted τ_3. Meanwhile, τ_4 on core 2 can execute because core 2 has been relinquished and τ_4 is not mutual-exclusive with τ_1 (recall that the scheduler is work-conserving). At time 6.5, τ_2 on core 3 arrives and it makes τ_4 suspended since τ_2 has higher-priority than τ_4. Core 2 is idling from time 6.5 to 7 due to co-runner locking. When τ_1 finishes execution at time 7, τ_3 resumes. All tasks finish execution before their deadlines. The time intervals affected by co-runner locking are shown with shaded areas (gray).

In summary, the priority-based control policy makes tasks in a mutual-exclusive relationship (ϵ_i) behave *as if they were preemptively scheduled on the same core based on their priorities*. Hence, we call such behavior as "co-runner preemption". This property helps schedulability of higher-priority tasks, especially when task priorities are assigned by RM or DM, because they are not blocked by lower-priority co-runner tasks.

Other Control Policies. We can also consider other execution control policies. In conventional real-time locking protocols for critical sections, first-come first-serve (FCFS) has been widely studied as an alternative to priority-based arbitration and has showed good properties to schedulability. Thus, we discuss the feasibility of FCFS for co-runner locking.

Fig. 2 illustrates the schedule of the same taskset as above under the FCFS-based control policy. Unlike the priority-based case, a task arrived earlier is prioritized when it comes to controlling the co-runner execution order. The first difference

can be seen at time 3 when τ_1 arrives. Although τ_1 has higher priority than τ_3, τ_3 continues execution due to FCFS. Then, τ_1 is blocked for 3 time units and misses the deadline. The same result happens to τ_2. As τ_4 has been already running, τ_2 is blocked until τ_4's completion and misses the deadline. Those blocking times by lower-priority tasks do not occur under the priority-based execution control. In fact, FCFS takes away the opportunity for higher-priority tasks to preempt lower-priority tasks, possibly resulting in poor schedulability.

One may also consider other orders. Under partitioned scheduling, comparing task priorities across cores may impose unfair scheduling penalty to particular cores because priorities were originally assigned to serve only the role of achieving per-core schedulability. This may become an issue especially when per-core taskset utilization is imbalanced or each core has tasks with a different range of deadlines, e.g., $[10, 100]$ ms on core 1 and $[100, 1000]$ ms on core 2. An arbitrary co-runner execution order separate from priority therefore has the potential to help such cases. However, this requires additional considerations; for example, to avoid deadlocks, total ordering must be ensured over priority and co-runner execution orders. We leave this as future work and focus on the priority-based policy in the rest of this paper.

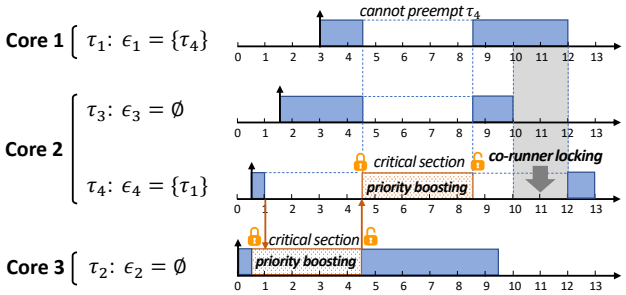
C. Implementation Considerations

The two rules of the execution control policy given in Sec. IV-B need to be implemented within the scheduler since they should be checked when a scheduling decision is made. The rule R1 should be placed before context-switching to the next task to run. R1 can also be applied after context-switching, but doing so would incur larger overhead as the task might need to be suspended immediately after the context switch. The rule R2 can be placed at the beginning of the scheduler, before R1 is assessed. A data structure for the co-runner exclusion set ϵ_i should be allocated for any task that is using co-runner locking, and this can be easily done by expanding the task control block (TCB). Other parameters we defined in the system model, e.g., S, G_i, ρ_i, are not required at runtime as they are used only for offline analysis purposes.

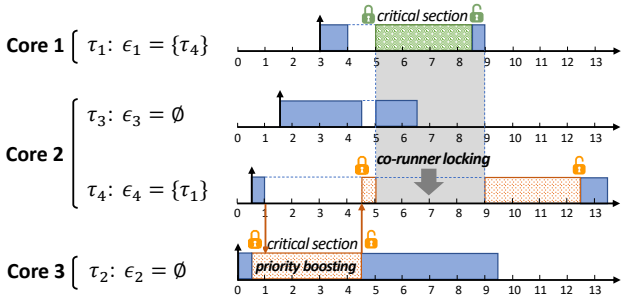
Co-runner locking requires changes to the scheduler, but the implementation difficulty is much lower than cache/DRAM bank partitioning which requires a detailed knowledge of address mapping in hardware and significant changes to page allocation in the virtual memory subsystem. If the system already has the core scheduling feature [1, 2, 7, 35] discussed in Sec. II, the implementation of co-runner locking can largely benefit from it. The runtime overhead to check co-runner conditions would be similar to PREM [44] which serializes the memory phases of all tasks on different cores at runtime. We will show the breakdown of this overhead in Sec. VII.

D. Tasks with Critical Sections

A question that can be raised is how co-runner locking interacts with conventional real-time synchronization protocols and how blocking time can be analyzed if tasks have critical sections. In multi-core systems, scheduling penalties caused by



(a) Case 1: τ_1 with no critical section



(b) Case 2: τ_1 with a critical section

Fig. 3: Co-runner locking with MPCP

critical sections are categorized into *local* and *remote blocking*. Local blocking is the delay for which a task needs to wait for lower-priority tasks on the same core. Remote blocking is the delay that a task has to wait due to the tasks on different cores. Remote blocking also causes additional delay due to back-to-back execution and multiple priority inversion, both of which are caused by self-suspending behavior [13, 34].

To analyze such delays, tasks with co-runner exclusion conditions can be treated as if they were on the same core. This is because under the priority-based execution control policy, suspension by higher-priority co-runner tasks has the same effect as the preemption by higher-priority tasks on the same core (called co-runner preemption). Hence, conventional multiprocessor synchronization protocols like MPCP [45] and FMLP [13] can be used with co-runner locking, with some extensions to the blocking time analysis. We do not provide a full analysis on this matter, but instead, we briefly discuss such extensions in the context of MPCP.

Let us consider a taskset $\Gamma = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ in a three-core system shown in Fig. 3a, where τ_1 and τ_4 are co-runner exclusive, i.e., $\epsilon_1 = \{\tau_4\}$ and $\epsilon_4 = \{\tau_1\}$, and τ_2 and τ_4 share the same mutex to access a global resource. If a higher-priority task with co-runner locking does not use a shared resource (τ_1 in Fig. 3a), the remote blocking time for a lower-priority task (τ_4) remains unaffected (same as without co-runner locking) because τ_1 cannot preempt τ_4 during critical section execution due to MPCP's priority boosting.[3] This also applies to the case where τ_1 uses the same shared resource as τ_4. However, τ_1 experiences priority-inversion (local) blocking from time 4.5 to 8.5, which can be analyzed by treating as if τ_1 were on the same core as τ_4.

[3]Other real-time multiprocessor synchronization protocols such as FMLP also have similar mechanisms to offer bounded remote blocking [13].

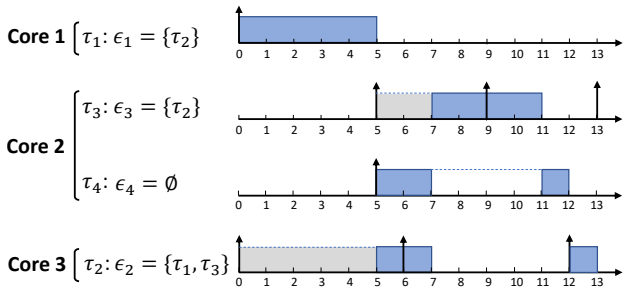


Fig. 4: Back-to-back execution effect due to co-runner locking

If a higher-priority task with co-runner locking uses a different shared resource (τ_1 in Fig. 3b), the remote blocking time for τ_4 is increased due to co-runner preemption (time 5.5 to 9). This can be bounded by capturing the critical section length of such higher-priority tasks as if they were accessing the same resource or executing on the same core.

Back-to-back execution due to self-suspending behavior is already considered by the co-runner locking analysis given in Sec. V, so no extra work is needed. Multiple priority inversions can happen because whenever a task suspends due to remote blocking, lower-priority tasks get a chance to request a mutex and will execute with priority boosting [34]. The number of priority inversions may increase due to co-runner locking since it causes additional suspension. But this can be bounded by the number of arrivals of higher-priority co-runners during the response time of the task under analysis.

The use of co-runner locking with real-time synchronization protocols therefore calls for a tradeoff between co-runner slowdown and critical section blocking time. This could be accommodated when determining co-runner exclusion sets (Sec. VI). In addition, timing penalties could be reduced by modeling slowdown factors of normal and critical-section segments separately (Sec. V-D). These are interesting extensions that can be built upon our work.

V. Schedulability Analysis

This section presents schedulability analysis for tasks under the co-runner locking scheme with priority-based execution control. We first show a baseline approach that can be obtained by directly extending the conventional iterative response-time test. To address the pessimism of the baseline, we then derive job-oriented and load-oriented slowdown analyses. Finally, we extend our task model to multi-segment tasks.

A. Baseline Analysis

Recall that the worst-case execution time (WCET) of a task τ_i can be upper-bounded by the worst-case execution requirement C_i multiplied by the maximum slowdown θ_i from true co-runners (Def. 7), i.e., $C_i \cdot \theta_i$. Another factor we need to consider for schedulability is the preemption-like delay caused by higher-priority tasks that execute on different cores but are in the co-runner exclusion set ϵ_i of τ_i. (Def. 4).

In addition, there is a back-to-back execution effect caused by higher-priority tasks on the same core which are suspended (preempted) by co-runners with even higher priority. Those co-runners themselves may also be delayed by the other co-runners. Fig. 4 gives an example. The two tasks τ_1 and τ_2 arrive together at time 0, but due to co-runner locking, τ_1 executes first and τ_2 is delayed until time 5. On core 2, τ_3 arrives at time 5 but τ_2 prevents τ_3's execution. Thus, τ_4 begins execution at time 5. The second job of τ_2 arrives at time 6, so τ_3 is further delayed until time 7. The task τ_3 executes its first job from time 7 to 9, and then executes its second job in a back-to-back manner. This *self-suspending behavior* of τ_3 causes larger than one job of interference to τ_4. However, τ_4 cannot directly see how this happens since τ_3's self-suspension is caused by τ_2's self-suspension which is caused by τ_1.

A simple solution can be derived by assuming that any higher-priority tasks, including those running on the same and different cores, are self-suspending tasks. Let us review the following lemma given by Bletsas et al. [11]:

Lemma 1 (from [11])**.** *The worst-case response time of a self-suspending task τ_i is upper bounded by:*

$$R_i = C_i + \sum_{\tau_j \in hpp(\tau_i)} \left\lceil \frac{R_i + (R_j - C_j)}{T_j} \right\rceil C_j \tag{1}$$

where $hpp(\tau_i)$ is the set of higher-priority tasks than τ_i on the same core.

This lemma captures the self-suspension behavior of an interfering task τ_j as a release jitter bounded by $R_j - C_j$. One can also use D_h instead of R_j in the summing term (1) [16]. Using this, we derive the following for our task model.

Lemma 2 (Baseline analysis)**.** *The worst-case response-time of a task τ_i under co-runner locking is upper bounded by:*

$$\begin{aligned} R_i = C_i \cdot \theta_i &+ \sum_{\tau_j \in hpp(\tau_i)} \left\lceil \frac{R_i + I_j(C_j \cdot \theta_j)}{T_j} \right\rceil C_j \cdot \theta_j \\ &+ \sum_{\tau_k \in \epsilon_i \wedge \tau_k \in hp(\tau_i)} \left\lceil \frac{R_i + I_k(C_k \cdot \theta_k)}{T_k} \right\rceil C_k \cdot \theta_k \end{aligned} \tag{2}$$

where

$$I_j(x) = \begin{cases} \max(R_j - x, 0) & , \exists \tau_y : \tau_y \in \epsilon_j \wedge \tau_y \in hp(\tau_j) \\ 0 & , otherwise \end{cases} \tag{3}$$

, and $hp(\tau_i)$ is the set of all higher-priority tasks than τ_i in the system. It can be solved by fixed-point iteration with the initial condition of $R_i = C_i \cdot \theta_i$. τ_i is schedulable if $R_i \leq D_i$.

Proof. Eq. (2) is an extension of Eq. (1). The first term is the execution time of τ_i, the second term bounds the preemption from higher-priority tasks on the same core, and the third term bounds the preemption by higher-priority tasks in the co-runner exclusion set ϵ_i. The self-suspending behavior of each higher-priority task τ_j is captured as a release jitter of $I_j(C_j \cdot \theta_j)$, where I_j returns $R_j - C_j \cdot \theta_j$ if τ_j has a co-runner exclusion relationship with τ_i, and zero, otherwise. □

B. Job-oriented Slowdown Analysis

In Eq. (2), the execution time of a task τ_j is simply captured by $C_j \cdot \theta_j$, assuming it gets the worst-case slowdown throughout its execution. However, this is overly pessimistic

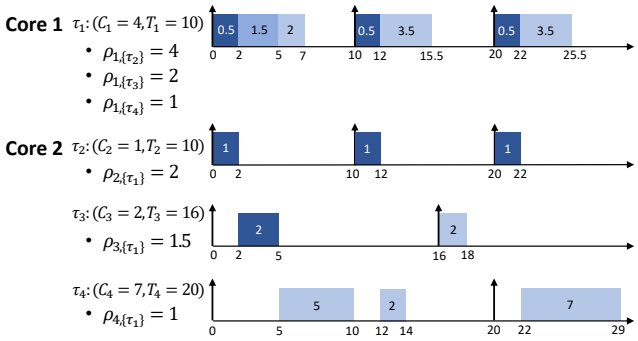


Fig. 5: Taskset 1

when the execution time of the worst-case co-runner task is short. Fig. 5 depicts an example where τ_1 runs on core 1 and the other three tasks run on core 2. The deadlines of these tasks are equal to their periods. The slowdown factor of each task for each co-runner set, $\rho_{i,s}$, is denoted in the figure, and the co-runner exclusion set ϵ_i is empty for all tasks. The number in each box represents the actual unit of execution progress made during that time, so the sum of these numbers for one period is equal to the task's execution requirement C_i. For instance, the first job of τ_1 is shown with three boxes: at first, it executes in parallel with τ_2 from time 0 to 2, experiencing the slowdown of 4 and making $2/4 = 0.5$ units of progress; secondly, it executes with τ_3 from time 2 to 5, making $3/2 = 1.5$ units of progress; and finally, its execution overlaps with τ_4 for 2 time units, making $2/1 = 2$ units of progress, and finishes at time 7. This clearly shows that the actual execution time of a task can be much smaller than what is bounded by Eq. (2), e.g., $4 \cdot 4 = 16$ for τ_1, thereby misjudging that τ_1 missed the deadline. To reduce this pessimism, we present an approach to obtain a tighter upper bound on the slowdown of each job.

The key to our job-oriented slowdown analysis lies in bounding how long interfering co-runners can execute during the response time of a task under analysis. We first analyze an upper bound on the cumulative execution time of one co-runner τ_k, $\zeta_{i,k}$, during the response time of τ_i, R_i. A simple approach is to capture the maximum number of occurrences of τ_k during R_i multiplied by the execution requirement C_k and the slowdown θ_k. Hence, it is given by:

$$\zeta_{i,k} = \min\left(\left(\left\lceil \frac{R_i}{T_k} \right\rceil + 1\right) C_k \cdot \theta_k, R_i\right) \tag{4}$$

where "+1" is to capture a carry-in job in the window of R_i. The reason for taking the minimum between the two terms is that the cumulative execution of τ_k during R_i cannot exceed R_i. However, since the use of the ceiling function may overestimate τ_k's job executing beyond R_i, Eq. (4) can be rewritten with a floor function as follows:

$$\begin{aligned}\zeta_{i,k} &= \min\left(\left(\left\lfloor \frac{R_i}{T_k} \right\rfloor + 1\right) C_k \cdot \theta_k + \min\left(R_i - \left\lfloor \frac{R_i}{T_k} \right\rfloor T_k, C_k \cdot \theta_k\right), R_i\right) \\ &= \min\left(\left(\left\lfloor \frac{R_i}{T_k} \right\rfloor + 1\right) C_k \cdot \theta_k + \min\left(R_i \bmod T_k, C_k \cdot \theta_k\right), R_i\right)\end{aligned} \tag{5}$$

On the other hand, Eq. (4) can also be improved by replacing the pessimistic "+1" term with a self-suspension release jitter [11, 16] (given in Eq. (3)):

$$\zeta_{i,k} = \min\left(\left\lceil \frac{R_i + I_k(C_k \cdot \theta_k)}{T_k} \right\rceil C_k \cdot \theta_k, R_i\right) \tag{6}$$

By combining these two improvements to Eq. (4), we can obtain the following.

Lemma 3. *The maximum cumulative execution time of a co-runner τ_k during the response time of τ_i is upper-bounded by*

$$\begin{aligned}\zeta_{i,k} = \min\Bigg(&\left\lfloor \frac{R_i + I_k(C_k \cdot \theta_k)}{T_k} \right\rfloor C_k \cdot \theta_k \\ &+ \min\Big((R_i + I_k(C_k \cdot \theta_k)) \bmod T_k, C_k \cdot \theta_k\Big), R_i\Bigg)\end{aligned} \tag{7}$$

Proof. Omitted as the steps are shown above. □

We now extend this to all tasks in a co-runner set.

Lemma 4. *The maximum cumulative time $\xi_{i,s}$ that all tasks in an non-empty co-runner set s ($s \neq \emptyset$) execute together in parallel during the response time of τ_i is defined as follows:*

$$\xi_{i,s} = \min_{\tau_j \in s} \zeta_{i,j} \tag{8}$$

Proof. Obviously, co-runner tasks in s cannot execute altogether longer than the execution time of the shortest one. Note that the back-to-back execution effect caused by tasks with self-suspending behavior is already bounded by $\zeta_{i,j}$. □

The above lemma is for non-empty co-runner sets $s \neq \emptyset$. For $s = \emptyset$, we define $\xi_{i,\emptyset} = \infty$. This might look counter-intuitive at a first glance, but indicates that the maximum cumulative time that other cores are idling is unbounded since there is no minimum execution time defined for each task.

Let us use $V_{i,k}$ to denote the k-th largest slowdown factor in the slowdown set ρ_i, and $X_{i,k}$ to denote the co-runner set corresponding to $V_{i,k}$.[4] Using these, we compute an upper bound on the execution time of a task τ_i as follows.

Theorem 1 (Job-oriented analysis)**.** *The execution time of a job of τ_i under co-runner locking is upper-bounded by*

$$C_i^* = \sum_{0<k\leq|\rho_i|} V_{i,k} \cdot \phi_{i,k} \tag{9}$$

where $\phi_{i,k}$ upper-bounds the maximum execution requirement of τ_i affected by a slowdown factor $V_{i,k}$ and is given by

$$\phi_{i,k} = \min\left(C_i - \sum_{0<l<k} \phi_{i,l}, \ \frac{\xi_{i,X_{i,k}}}{V_{i,k}}\right) \tag{10}$$

Proof. To maximize the execution time of τ_i, it has to get the largest slowdown factor $V_{i,1}$ ($k = 1$) for as much execution requirement as possible. On the one hand, this is bounded by τ_i's execution requirement C_i; hence, $\phi_{i,1} \leq C_i$ and the execution time $C_i^* \leq \phi_{i,1} \cdot V_{i,1}$. On the other hand, the co-runner set corresponding to $V_{i,1}$ cannot execute in parallel with τ_i longer than $\xi_{i,1}$, and the maximum execution requirement that

[4] $V_{i,k}$ is ordered in decreasing order of slowdown factors (i.e., $V_{i,1}$ is the largest), but the number of co-runners in $X_{i,k}$ may not increase with k (e.g., it is possible that $|X_{i,1}| < |X_{i,2}|$). In other words, we do not assume that the slowdown of τ_i is just a function of the number of co-runners.

τ_i can accomplish during this time is $\xi_{i,X_{i,1}}/V_{i,1}$. Therefore, an upper bound on τ_i's execution requirement affected by $V_{i,1}$ is obtained by taking the minimum between these two, i.e., $\phi_{i,1} = \min(C_i, \xi_{i,X_{i,1}}/V_{i,1})$.

If $C_i \leq \xi_{i,X_{i,1}}/V_{i,1}$, the execution time of τ_i is bounded by $C_i \cdot V_{i,1}$. Otherwise, τ_i may experience slowdown from other co-runner sets ($k \geq 2$) for its remaining execution requirement, $C_i - \xi_{i,X_{i,1}}/V_{i,1} = C_i - \phi_{i,1}$.

For $k = 2$, the execution time of τ_i is the sum of $V_{i,1} \cdot \phi_{i,1}$ and the time required to complete the remaining execution requirement of $C_i - \phi_{i,1}$. Hence, $V_{i,2}$ applies to the minimum of $C_i - \phi_{i,1}$ (whole remaining part) and $\xi_{i,X_{i,2}}/V_{i,2}$ (progress achievable during $V_{i,2}$), which is given by $\phi_{i,2}$. If there still remains non-zero execution requirement, this sequence continues for the next largest slowdown factor.

For $k = |\rho_i|$, the least slowdown factor of 1 applies because the co-runner set $X_{i,|\rho_i|}$ is $\emptyset$. Hence, any remaining execution requirement at this point can be executed with no slowdown, and $\phi_{i,|\rho_i|}$ ensures this by taking the minimum between the remaining execution requirement and $\xi_{i,\emptyset} = \infty$. The steps for all co-runner sets from $k = 1$ to $|\rho_i|$ are represented by the summing term in Eq. (9). It is worth noting that Eq. (10) cannot be negative because it considers the remaining execution requirement of τ_i and $\sum \phi_{i,k}$ cannot exceed C_i. □

Using this theorem, the response time test given in Eq. (2) can be rewritten as follows:

$$R_i = C_i^* + \sum_{\tau_j \in hpp(\tau_i)} \left\lceil \frac{R_i + I_j(C_j^*)}{T_j} \right\rceil C_j^* + \sum_{\tau_k \in \epsilon_i \wedge \tau_k \in hp(\tau_i)} \left\lceil \frac{R_i + I_k(C_k^*)}{T_k} \right\rceil C_k^* \tag{11}$$

where the initial value for fixed-point iteration is $R_i = C_i$ (not C_i^*, since C_i^* depends on R_i).

Example. For the taskset in Fig. 5, the job-oriented slowdown analysis bounds the worst-case response time of each task as follows: $R_1 = 7$, $R_2 = 2$, $R_3 = 5$, and $R_4 = 14$, which exactly match the timeline shown in the figure. However, the job-oriented approach can be pessimistic for a different type of tasksets. Let us consider another example given in Fig. 6. The figure depicts the worst case, i.e., shifting around the jobs of τ_1 and τ_2 does not increase τ_3's response time. Our focus is to analyze the response time of τ_3, which is shown to be the same as its period, 16. However, the job-oriented analysis determines τ_3 fails to meet the deadline. This is because it assumes that τ_1 is slowing down both τ_2's job and τ_3's job for 2 time units each, but that cannot occur in reality.

C. Load-oriented Slowdown Analysis

The job-oriented analysis computes each job's slowdown separately and then analyzes the response time. Alternatively, we can directly analyze the response time of a task τ_i by iteratively capturing the cumulative slowdown imposed on all the *load* during τ_i's response time. Here, the load means the execution requirement of the task itself and those of other

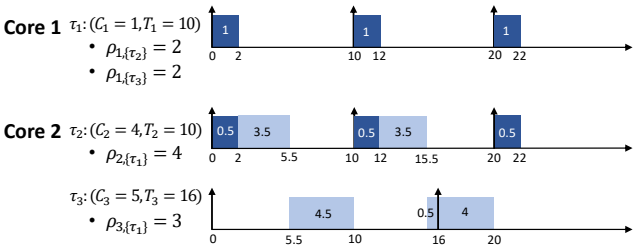


Fig. 6: Taskset 2

higher-priority tasks preempting that task. Below we define additional terms required for this load-oriented approach.

Def. 8. *H_i is the set of tasks directly contributing to the load for τ_i's response time. This includes τ_i, higher-priority tasks on the same core, and higher-priority mutually-exclusive co-runner tasks.*

$$H_i = \{\tau_j \mid \tau_j = \tau_i \vee \tau_j \in hpp(\tau_i) \vee (\tau_j \in \epsilon_i \wedge \tau_j \in hp(\tau_i))\}$$

Based on H_i, the load E_i can be defined as follows.

Def. 9 (Load). *E_i is the sum of all execution requirements of tasks in H_i during τ_i's response time R_i. Note that $I_j(C_j)$ captures the self-suspending behavior of τ_j as in Eq. (2).*

$$E_i = C_i + \sum_{\tau_j \in H_i \setminus \{\tau_i\}} \left\lceil \frac{R_i + I_j(C_j)}{T_j} \right\rceil C_j \tag{12}$$

We now introduce G_i^* and ρ_i^*, which are similar to G_i (Def. 5) and ρ_i (Def. 6) but different in the sense that the new parameters are defined for the tasks in H_i.

Def. 10. *G_i^* is the set of "true" co-runner sets of tasks in H_i.*

$$G_i^* = \{s \mid \exists \tau_j (\tau_j \in H_i) \wedge (s \in G_j)\}$$

Note that G_i^* is a set so there is no duplicate element.

Def. 11. *ρ_i^* is the set of slowdown factors corresponding to the co-runner sets in G_i^*. If there are multiple slowdown factors for the same co-runner set, ρ_i^* stores only the maximum.*

$$\rho_i^* = \{x \mid \exists s (s \in G_i^*) \wedge (x = \max_{\tau_j \in H_i} \sigma_{j,s})\}$$

Let us use $V_{i,k}^*$ to denote the k-th largest slowdown factor in the modified slowdown set ρ_i^*, and $X_{i,k}^*$ to denote the co-runner set corresponding to $V_{i,k}^*$. We now derive an upper bound on the worst-case response time of a task τ_i as follows.

Theorem 2 (Load-oriented analysis)**.** *The worst-case response time of a task τ_i in the presence of co-runner interference is upper-bounded by*

$$R_i = \sum_{0 < k \leq |\rho_i^*|} V_{i,k}^* \cdot \phi_{i,k}^* \tag{13}$$

where

$$\phi_{i,k}^* = \min\left(E_i - \sum_{0 < l < k} \phi_{i,l}^*, \ \frac{\xi_{i,X_{i,k}^*}}{V_{i,k}^*}\right) \tag{14}$$

Note that this is a recurrence since E_i depends on the previous value of R_i. It can be solved by fixed-point iteration with $R_i = \sum_{\tau_j \in H_i} C_j$ for the initial value of E_i.

Proof. Eq. (13) is analogous to Eq. (9) and can be proved in the same way as in Theorem 1. □

Example. The load-oriented analysis computes the worst-case response time of the tasks in Fig. 6 as follows: $R_1 = 2$, $R_2 = 5.5$, and $R_3 = 16$. In fact, this taskset is favorable to the load-oriented analysis because, when analyzing τ_3, the interfering co-runner τ_1 executes for only 4 time units during R_3. It is worth noting that the job-oriented and the load-oriented analyses do not dominate each other (e.g., the load-oriented analysis declares a failure to τ_4 in Fig. 5). Given that both analyses produce upper bounds on the worst-case response time, we propose to use them jointly, by taking the minimum between the two.

D. Extension to Multi-segment Tasks

We now extend our task model such that each task consists of a sequence of one or more execution segments. This multi-segment task model is effective in capturing different slowdown factors for each part of a program and has been used in prior work on co-runner dependent execution time [6]. A multi-segment task $\boldsymbol{\tau}_i$ can be represented as a collection of single-segment tasks, all of which have the same period, deadline, and priority: $\boldsymbol{\tau}_i := (\tau_{i_1}, \tau_{i_2}, ..., \tau_{i_n})$, where τ_{i_1} and τ_{i_n} are the first and the last segment of $\boldsymbol{\tau}_i$. For schedulability analysis, the response time tests provided in earlier subsections can be reused with minor modifications: (i) the execution time of a multi-segment task under analysis needs to be captured as the sum of execution times of tasks corresponding to its segments, and (ii) the segments of higher-priority multi-segment tasks can be treated as independent single-segment tasks because our analysis is agnostic to their execution order but captures how long they execute. Below we present how each analysis can be extended.

Baseline analysis. The response time of a multi-segment task $\boldsymbol{\tau}_i$ is upper-bounded by:

$$R_i = \mathbf{C}_i + \sum_{\tau_j \in hpp(\boldsymbol{\tau}_i)} \left\lceil \frac{R_i + I_j(C_j \cdot \theta_j)}{T_j} \right\rceil C_j \cdot \theta_j + \sum_{\tau_k \in \boldsymbol{\varepsilon}_i \wedge \tau_k \in hp(\boldsymbol{\tau}_i)} \left\lceil \frac{R_i + I_k(C_k \cdot \theta_k)}{T_k} \right\rceil C_k \cdot \theta_k \tag{15}$$

where $\mathbf{C}_i = \sum_{\tau_{i'} \in \boldsymbol{\tau}_i} C_{i'} \cdot \theta_{i'}$, $\boldsymbol{\varepsilon}_i = \cup_{\tau_{i'} \in \boldsymbol{\tau}_i} \epsilon_{i'}$, and the initial condition for fixed-point iteration is $R_i = \mathbf{C}_i$.

Job-oriented slowdown analysis. The response time test given in Eq. (11) can be rewritten for multi-segment tasks:

$$R_i = \mathbf{C}_i^* + \sum_{\tau_j \in hpp(\boldsymbol{\tau}_i)} \left\lceil \frac{R_i + I_j(C_j^*)}{T_j} \right\rceil C_j^* + \sum_{\tau_k \in \boldsymbol{\varepsilon}_i \wedge \tau_k \in hp(\boldsymbol{\tau}_i)} \left\lceil \frac{R_i + I_k(C_k^*)}{T_k} \right\rceil C_k^* \tag{16}$$

where $\mathbf{C}_i^* = \sum_{\tau_{i'} \in \boldsymbol{\tau}_i} C_{i'}^*$ and the initial condition for fixed-point iteration is $R_i = \sum_{\tau_{i'} \in \boldsymbol{\tau}_i} C_{i'}$.

Load-oriented slowdown analysis. For the load-oriented analysis, we need to redefine two of the parameters to take into account all segments of $\boldsymbol{\tau}_i$. First, H_i, the set of tasks contributing to the load for $\boldsymbol{\tau}_i$'s response time is changed to:

$$H_i = \{\tau_j \mid \tau_j \in \boldsymbol{\tau}_i \vee \tau_j \in hpp(\boldsymbol{\tau}_i) \vee (\tau_j \in \boldsymbol{\varepsilon}_i \wedge \tau_j \in hp(\boldsymbol{\tau}_i))\}$$

Secondly, E_i, the sum of all execution requirements of tasks in H_i during R_i is changed to:

$$E_i = \sum_{\tau_{i'} \in \boldsymbol{\tau}_i} C_{i'} + \sum_{\tau_j \in H_i \setminus \boldsymbol{\tau}_i} \left\lceil \frac{R_i + I_j(C_j)}{T_j} \right\rceil C_j \tag{17}$$

With these H_i and E_i, the response time test given in Eq. (13) can be used for analyzing multi-segment tasks.

Algorithm 1 Simulated Annealing (SA)

Require: Γ: a taskset, α: cooling rate, t_{min}: minimum temperature threshold, iter: # of iterations per temperature, time_limit: maximum time for search

```
/* Sanity check: no need to run if already schedulable */
Run response-time test for all tasks
if taskset is schedulable then
    return success
t = 1.0 /* t: current temperature */
n_prev = # of schedulable tasks w/o co-runner locking
while t > t_min do
    for k = 1 to iter do
        Randomly picks two tasks, τ_i and τ_j
        if τ_j ∉ ε_i then
            ε_i = ε_i ∪ {τ_j}; ε_j = ε_j ∪ {τ_i}
        else
            ε_i = ε_i \ {τ_j}; ε_j = ε_j \ {τ_i}
        Update co-runner sets G and slowdown factors ρ
        Run response-time test for all tasks
        n_cur = # of schedulable tasks
        if taskset is schedulable then
            return success
        /* probabilistic acceptance */
        if exp((n_cur − n_prev)/|Γ|/t) > random(0, 1) then
            n_prev = n_cur
        else
            Revert ε_i and ε_j
            Update co-runner sets G and slowdown factors ρ
    t = t · α
    if elapsed_time > time_limit then
        break
return failure
```

VI. Co-runner Exclusion Algorithms

Choosing the right set of tasks for a co-runner exclusion set ϵ_i is critical to achieving schedulability. However, there is no clear strategy that always leads to good results. Adding a co-runner task to ϵ_i may reduce slowdown-induced delay but may increase preemption-induced delay to others. Exhaustively searching the entire solution space is not feasible even for a reasonable-size system. Motivated by these difficulties, we develop two heuristic algorithms, the one based on simulated annealing and the other based on a simple intuition.

A. Simulated Annealing

We first formulate the problem of finding a co-runner exclusion set ϵ_i into simulated annealing (SA) as shown in Alg. 1. It takes as input a taskset Γ and other SA-related parameters including the cooling rate α, the minimum temperature threshold t_{min}, the number of iterations per temperature,

Algorithm 2 Maximize Relative Slack (MaxSlack)

Require: Γ: a taskset
1: /* Sanity check: no need to run if already schedulable */
2: Run response-time test for all tasks
3: **if** taskset is schedulable **then**
4: **return success**
5: **for all** $\tau_i \in \Gamma$ in descending order of priority **do**
6: /* Consider all co-runner tasks τ_k of τ_i */
7: **for all** $\tau_k \in \Gamma \wedge \tau_k \neq \tau_i$ in descending order of priority **do**
8: /* Check if τ_k and τ_i are already mutually-exclusive */
9: **if** $\tau_k \in \epsilon_i$ **then**
10: **continue**
11: /* Check slack before/after τ_k is selected for exclusion */
12: $slack_{old} = \sum_{\forall \tau_j} \frac{D_j - \min(R_j, D_j)}{T_j}$
13: $\epsilon_i = \epsilon_i \cup \{\tau_k\}$; $\epsilon_k = \epsilon_k \cup \{\tau_i\}$;
14: Update co-runner sets G and slowdown factors ρ
15: Run response-time test for all tasks
16: $slack_{new} = \sum_{\forall \tau_j} \frac{D_j - \min(R_j, D_j)}{T_j}$
17: **if** $slack_{new} < slack_{old}$ **then**
18: /* Discard if resulting slack is smaller than before */
19: Revert ϵ_i and ϵ_j
20: Update co-runner sets G and slowdown factors ρ
21: **if** taskset is schedulable **then**
22: **return success**
23: **return failure**

and the time limit. Since our goal is taskset schedulability, the cost for SA is determined by the number of schedulable tasks in a given taskset. The variables to change are ϵ_i. For each temperature level, the algorithm first generate a new random set of variables by randomly picking two tasks and toggling their co-runner exclusion relationship (lines 9-13). Then it updates the set of true co-runner sets and the set of slowdown factors for both tasks, i.e., G_i, G_j, ρ_i, and ρ_j, and checks the schedulability by running the response-time test. If the taskset is schedulable, the algorithm returns success. Otherwise, it accepts the new variable set with a probability computed by $\exp()$ (line 20); if not acceptable, then the variable set is reverted to the previous one. This continues until the temperature t cools down below the threshold t_{min} or it hits the time limit. If the algorithm cannot make the taskset schedulable until this point, it returns failure.

B. Maximizing Relative Slack

Our second heuristic focuses on the simple fact that increasing the slack of higher-priority tasks can help improve the schedulability of lower-priority tasks. Alg. 2 depicts our algorithm, MaxSlack, which aims to maximize relative slack. Basically, it iterates over all tasks in descending order of priority (so the highest-priority task first) and adds a co-runner task to the exclusion set ϵ if doing so increases the cumulative sum of slack for all tasks. Specifically, for each τ_i, the algorithm finds all co-runner tasks τ_k of the task τ_i. If τ_i and τ_k are not already in a mutually-exclusive relationship, the algorithm checks if the sum of relative slack after adding τ_k to ϵ_i and τ_i to ϵ_k is greater than before (lines 12 and 16). The sum of relative slack for all tasks is computed by $\sum_{\forall \tau_j} \frac{D_j - \min(R_j, D_j)}{T_j}$, where $\min(R_j, D_j)$ is to consider a task whose response time is not bounded. If the new slack value is smaller than before, ϵ_i and ϵ_k are reverted. Otherwise, the algorithm keeps these co-runner exclusion sets and moves on to the next highest-priority task.

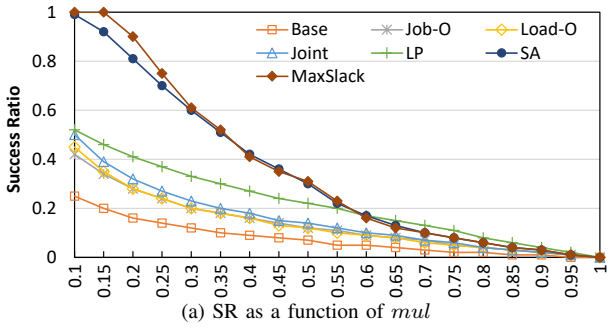
VII. Evaluation

A. Schedulability Tests

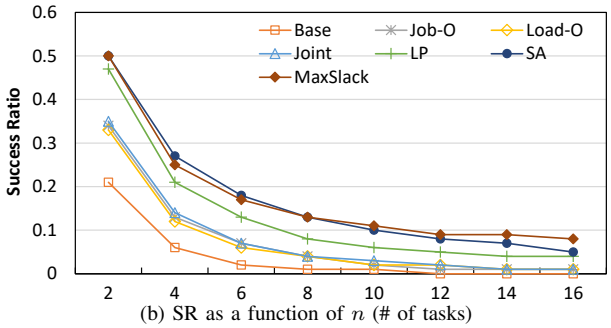
We evaluate the performance of our response-time analysis and the effects of co-runner locking in taskset schedulability by using random tasksets. We follow the taskset generation procedure in [6] to generate co-runner dependent execution time and use the same parameters as in that paper. Below we summarize this procedure. For each taskset, we first generate n tasks and assign priorities based on the Deadline Monotonic (DM) policy. Tasks are allocated to M CPU cores based on the worst-fit decreasing (WFD) heuristic in order to balance the workload across cores. For WFD, tasks are ordered based on task utilization $U_i = C_i/T_i$, which is the case where a task does not experience co-runner slowdown. Once tasks are assigned to cores, we scale up the execution requirements of all tasks by the rate of 1.01 at a time while the taskset remains schedulable, assuming there is no co-runner interference. Hence, after this procedure, multiplying each task's execution requirement by 1.01 makes the taskset unschedulable. Then, the execution requirements of all tasks are multiplied by an experiment control parameter $mul \leq 1$. This mul value serves as an indicator for per-core utilization. For co-runner slowdown, we generate the entire set of co-runner sets, and randomly choose a slowdown value from $[1, 1/progmin]$ for each co-runner set s, where $progmin \leq 1$ is another control parameter. Once generated, co-runner slowdown values are reordered such that if $s \subset s'$, then $\sigma_{i,s} \leq \sigma_{i,s'}$. This reordering is to reflect the reality that a larger number of co-running tasks tend to cause a greater slowdown.

Based on the above procedure, a total of 2,128,000 tasksets are generated with the combinations of the following parameters which are obtained from [6]: the number of tasks $n \in \{2, 4, 6, 8, 10, 12, 14, 16\}$, the number of segments per task $\in \{1, 2\}$, the number of CPU cores $M \in \{2, 3, 4, 5, 6, 7, 8\}$, $mul \in \{0.05, 0.10, 0.15, ..., 0.95, 1.00\}$, and $progmin \in \{0.05, 0.10, 0.15, ..., 0.95, 1.00\}$ and $progmin < mul$.

We use these tasksets to assess the performance of the response-time tests presented in Sec. V. Seven methods are compared: (i) our baseline analysis (Base), (ii) job-oriented analysis (Job-O), (iii) load-oriented analysis (Load-O), (iv) joint analysis that takes a minimum between job-oriented and load-oriented analysis results for the response time of each task (Joint), (v) the method proposed in [6] that formulates the schedulability analysis of tasks with co-runner dependent execution time into linear programming (LP), (vi) joint analysis with simulated annealing for finding co-runner exclusion sets (SA), (vii) joint analysis with the MaxSlack algorithm (MaxSlack). The first five methods, Base, Job-O, Load-O, Joint, LP, do not use co-runner locking (i.e., $\epsilon_i = \emptyset$ for all tasks). LP is the only existing work that can be directly compared with our work; however, LP does not support co-runner locking and it is not trivial to extend it. In case of SA,



(a) SR as a function of mul



(b) SR as a function of n (# of tasks)

Fig. 7: Schedulability success ratio (SR)

TABLE I: Algorithm running time per taskset on an x86@2.35 GHz processor

Name	Avg (ms)	Max (ms)	Name	Avg (ms)	Max (ms)
SA	1189.4	30391.1	MaxSlack	54.4	2108.2

we used the time limit of 30s, iter $= 30$, $t_{min} = 0.001$, and $\alpha = 0.99$.

Fig. 7 shows the schedulability success rate of the seven different methods. Each sub-figure demonstrates the schedulability success rate of each method based on different criteria (mul and # of tasks). Among the methods that do not use co-runner locking, both Job-O and Load-O outperform Base by a noticeable amount, and Joint performs better than these two. It is also shown that Job-O and Load-O do not dominate each other. LP performs the best among those that do not use co-runner locking. However, the analytical benefit of LP comes at a high computational cost. While Job-O, Load-O, and Joint take only less than 10 ms per taskset for a system with 8 cores, LP takes more than 100 seconds per taskset on overage for the same system. Such a high cost makes it difficult to be used with the algorithms to find co-runner exclusion sets because they check schedulability iteratively until a solution is found.

As can be seen from SA and MaxSlack, the use of co-runner locking brings significant benefit in schedulability. Specifically, MaxSlack achieves 100% of success ratio for $mul \leq 0.15$, and almost $2\times$ of improvement compared to the "no locking" methods until mul exceeds 0.3. Another interesting thing is that MaxSlack yields better results than SA in many cases, despite its simplicity and shorter running time (see Table I). There are some cases where LP performs slightly better than MaxSlack and SA ($0.65 \leq mul$), but this is due to the pessimism of our schedulability analysis when the utilization is high.

TABLE II: Implementation cost

Name	Avg (μs)	Max (μs)
Check other tasks in ϵ_i	0.29	9.63
Send suspend signal	0.45	8.99
Suspend running task	20.13	120.06
Wake up task	5.53	50.97

TABLE III: Taskset information for case study

Task	C_i ms	T_i ms	CPU	Type	Slowdown factors (selected)
τ_1	90.4	150	CPU1	Mem	$\sigma_{1,\{\tau_2,\tau_3,\tau_5\}} = 1.72$ $\sigma_{1,\{\tau_3,\tau_5,\tau_6\}} = 1.81$
τ_2	20.0	200	CPU2	Comp	$\forall s \in S_2 : \sigma_{2,s} = 1.01$
τ_3	90.4	300	CPU0	Mem	$\sigma_{3,\{\tau_1,\tau_2,\tau_5\}} = 1.72$ $\sigma_{3,\{\tau_1,\tau_5,\tau_6\}} = 1.81$ $\sigma_{3,\{\tau_2,\tau_4,\tau_5\}} = 1.12$ $\sigma_{3,\{\tau_4,\tau_5,\tau_6\}} = 1.18$
τ_4	100.0	400	CPU1	Comp	$\forall s \in S_4 : \sigma_{4,s} = 1.01$
τ_5	90.4	480	CPU3	Mem	$\sigma_{5,\{\tau_1,\tau_2,\tau_3\}} = 1.18$ $\sigma_{5,\{\tau_1,\tau_3,\tau_6\}} = 1.81$ $\sigma_{5,\{\tau_2,\tau_3,\tau_4\}} = 1.12$ $\sigma_{5,\{\tau_3,\tau_4,\tau_6\}} = 1.72$
τ_6	90.4	600	CPU2	Mem	$\sigma_{6,\{\tau_1,\tau_3,\tau_5\}} = 1.81$ $\sigma_{6,\{\tau_3,\tau_4,\tau_5\}} = 1.72$

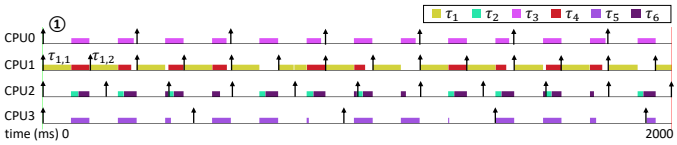
B. Case Study

We implemented the co-runner locking scheme in the Linux kernel v4.9.140 running on an Nvidia Xavier AGX platform. The implementation was done following our discussion in Sec. IV-C. The Xavier board has a total of 8 CPU cores and provides multiple power modes. We used the MAXN power mode (2.2 GHz) and disabled dynamic frequency scaling to minimize performance variability.

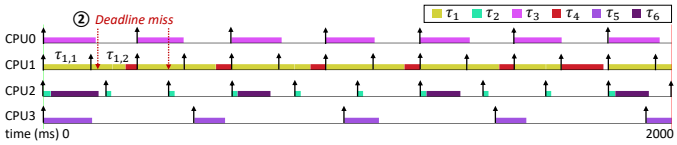
Overhead. Co-runner locking incurs the following overhead at runtime: (i) checking co-runner conditions (ϵ_i), (ii) sending a suspension request to another core if a lower-priority co-runner has to be suspended, (iii) suspending a currently-running task upon request, and (iv) waking up a task from the waiting list if there is any. Item 1 occurs whenever a task with $\epsilon_i \neq \emptyset$ enters the scheduler, but the others happen only under specified conditions. Table III shows the overhead observed from our implementation. Item 1 has very low overhead since it merely requires reading other tasks' running states. While the frequency of other items varies by tasksets and ϵ_i, we consider this cost is acceptable and can be optimized.

Taskset. Table III summarizes the taskset used in our case study. Tasks are ordered in descending priority, i.e., τ_1 is the highest-priority task. Memory-intensive tasks (τ_1, τ_3, τ_5, and τ_6) are based on the `latency` program [62] and their performance is highly affected by inter-core cache interference. In contrast, CPU-intensive tasks (τ_2 and τ_4) have little slowdowns from co-running tasks (i.e., $\forall s : \sigma_{2,s} = \sigma_{4,s} = 1.01$). The tasks are assigned to CPU0-3 of the Xavier board. The slowdown factors of the tasks are estimated by the measurement-based method in [6], and only some of them are depicted in the table for space reasons. Note that these slowdowns ($\leq 1.81\times$) are much smaller than the worst cases reported in the literature, e.g., $> 5\times$ with cache and DRAM bank partitioning in a quad-core system [63], and thus less favorable for co-runner locking.

Fig. 8 shows the execution traces of the taskset for 2s with and without co-runner locking. We recorded the traces using `trace-cmd` and `kernelshark`, and then annotated



(a) Scheduling with co-runner locking



(b) Scheduling without co-runner locking

Fig. 8: Taskset execution traces

with arrows to indicate the release time of the highest-priority task on each CPU. All the tasks were released together at time 0. In case of co-runner locking, only τ_1 is determined to run exclusively from other co-runners by MaxSlack (Alg. 2), i.e., $\epsilon_1 = \{\tau_2, \tau_3, \tau_5, \tau_6\}$ and $\epsilon_2 = \epsilon_3 = \epsilon_5 = \epsilon_6 = \{\tau_1\}$. Let us compare the results from the two cases. First, in Fig. 8a, τ_1 does not experience any inter-core interference since no co-runner exists during its execution; hence, all tasks in the system meet their deadlines. Second, in Fig. 8b, the first instance of τ_1, $\tau_{1,1}$, executes beyond its deadline due to the slowdown from co-runners. This delays the start time of the second instance, $\tau_{1,2}$, which in turn causes $\tau_{4,1}$ to miss its deadline too. The execution times of the memory-intensive tasks, τ_3, τ_5, and τ_6, are also much longer than those with co-runner locking. While this taskset could also be made schedulable by existing resource partitioning [54, 63] or interference-aware allocation [42, 59], the results show the potential of co-runner locking as an alternative approach.

VIII. Conclusion

In this paper, we presented the co-runner locking scheme to address the multi-core timing interference problem. Unlike existing approaches, our scheme takes a task synchronization approach to prevent some co-runner tasks from executing in parallel if they cause a large slowdown. The details on co-runner locking properties and runtime control policies were discussed. For the selection of mutually-exclusive co-runner sets, we developed two heuristic algorithms, simulated annealing and maximizing relative slack. Experimental results with randomly generated tasksets show that our analysis is much less pessimistic compared to the baseline approach. Case study results demonstrate that co-runner locking is effective even when inter-core interference is moderate. Therefore, we believe that co-runner locking can serve as an effective alternative to address the "one-out-of-m" problem when resource partitioning methods are unavailable or when they cannot eliminate all the interference penalties.

Acknowledgment

Copyright 2021 Carnegie Mellon University and IEEE. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. DM21-0735

References

[1] VMware response to L1 Terminal Fault - VMM (L1TF - VMM) speculative-execution vulnerability in Intel processors for vSphere: CVE-2018-3646 (55806). https://kb.vmware.com/s/article/55806. Accessed: Sep. 2021.

[2] What's new in Xen 4.13. https://xenproject.org/2019/12/18/whats-new-in-xen-4-13/. Accessed: Sep. 2021.

[3] AbsInt. aiT WCET Analyzer. https://www.absint.com/ait/.

[4] W. Ali, R. Pellizzoni, and H. Yun. Virtual gang scheduling of parallel real-time tasks. In *Design, Automation Test in Europe Conference (DATE)*, 2021.

[5] W. Ali and H. Yun. RT-Gang: Real-time gang scheduling framework for safety-critical systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[6] B. Andersson, H. Kim, D. de Niz, M. Klein, R. R. Rajkumar, and J. Lehoczky. Schedulability analysis of tasks with corunner-dependent execution times. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(3):71, 2018.

[7] B. Armstrong et al. Managing Hyper-V hypervisor scheduler types. https://docs.microsoft.com/en-us/windows-server/virtualization/hyper-v/manage/manage-hyper-v-scheduler-types. Accessed: Sep. 2021.

[8] M. A. Awan, K. Bletsas, P. F. Souto, B. Akesson, and E. Tovar. Mixed-criticality scheduling with dynamic redistribution of shared cache. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2017.

[9] M. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[10] M. Becker, D. Dasari, B. Nicolic, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[11] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, 2015.

[12] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *IEEE Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2007.

[13] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-time Operating Systems*. PhD thesis, Chapel Hill, NC, USA, 2011.

[14] B. B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[15] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *Euromicro conference on real-time systems*, 2009.

[16] J.-J. Chen et al. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.

[17] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2011.

[18] G. Elliott, B. C. Ward, J. H. Anderson, et al. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[19] P. Gai et al. A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2003.

[20] S. Hosseinimotlagh and H. Kim. Thermal-aware servers for real-time tasks on multi-core GPU-integrated embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[21] P. Huang, G. Giannopoulou, R. Ahmed, D. B. Bartolini, and L. Thiele. An isolation scheduling model for multicores. In *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[22] D. Iorga et al. Slow and steady: Measuring and tuning multicore interference. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020.

[23] X. Jin, H. Chen, X. Wang, Z. Wang, X. Wen, Y. Luo, and X. Li. A simple cache partitioning approach in a virtualized environment. In *IEEE Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2009.

[24] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding and reducing memory interference in COTS-based multi-core systems. *Real-Time Systems*, 52(3):356–395, 2016.

[25] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. R. Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[26] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[27] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access with improved analysis. *Journal of Systems Architecture*, 88:97–109, 2018.

[28] H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *ACM International Conference on Embedded Software (EMSOFT)*, 2016.

[29] H. Kim and R. Rajkumar. Predictable shared cache management for multi-core real-time virtualization. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(1):1–27, 2017.

[30] H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *IEEE Real-Time Systems Symposium (RTSS)*, 2014.

[31] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[32] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[33] P. Kocher et al. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (SP)*, 2019.

[34] K. Lakshmanan, D. d. Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[35] M. Larabel. Core scheduling looks like it will be ready for Linux 5.14 to avoid disabling SMT/HT. https://www.phoronix.com/scan.php?page=news_item&px=Core-Scheduling-Linux-5.14. Accessed: Sep. 2021.

[36] Y. Lim and H. Kim. Cache-aware real-time virtualization for clustered multi-core platforms. *IEEE Access*, 7:128628–128640, 2019.

[37] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[38] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2013.

[39] C. E. Nemitz, K. Yang, M. Yang, P. Ekberg, and J. H. Anderson. Multiprocessor real-time locking protocols for replicated resources. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.

[40] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2014.

[41] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *International Symposium on Computer Architecture (ISCA)*, 2009.

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

[43] P. Patel et al. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2018.

[44] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2011.

[45] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *IEEE Real-Time Systems Symposium (RTSS)*, 1988.

[46] Certification Authorities Software Team (CAST). Position Paper CAST-32A Multi-core Processors. https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/media/cast-32A.pdf, 2016.

[47] H. Rihani, M. Moy, C. Maiza, R. I. Davis, and S. Altmeyer. Response time analysis of synchronous data flow programs on a many-core processor. In *International Conference on Real-Time Networks and Systems (RTNS)*, 2016.

[48] B. Rouxel, S. Derrien, and I. Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–20, 2017.

[49] S. Saha et al. STGM: Spatio-temporal GPU management for real-time tasks. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2019.

[50] S. Skalistis and A. Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[51] S. Skalistis and A. Kritikakou. Dynamic interference-sensitive run-time adaptation of time-triggered schedules. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[52] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso. E-WarP: A system-wide framework for memory bandwidth profiling and management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[53] M. R. Soliman and R. Pellizzoni. PREM-based optimal task segmentation under fixed priority scheduling. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2019.

[54] N. Suzuki et al. Coordinated bank and cache coloring for temporal protection of memory accesses. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, 2013.

[55] B. C. Ward. Relaxing resource-sharing constraints for improved hardware management and schedulability. In *IEEE Real-Time Systems Symposium (RTSS)*, 2015.

[56] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson. Making shared caches more predictable on multicore platforms. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[57] M. Xu, L. T. Phan, I. Lee, O. Sokolsky, S. Xi, C. Lu, and C. Gill. Cache-aware compositional analysis of real-time multicore virtualization platforms. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[58] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[59] M. Xu, L. T. X. Phan, H.-Y. Choi, Y. Lin, H. Li, C. Lu, and I. Lee. Holistic resource allocation for multicore real-time systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019.

[60] M. Xu, L. Thi, X. Phan, H.-Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[61] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 65(9):2739–2751, 2015.

[62] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 2014.

[63] H. Yun, R. Pellizzon, and P. K. Valsan. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[64] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.