Simultaneous Multithreading in Mixed-Criticality Real-Time Systems

Joshua Bakita, Shareef Ahmed, Sims Hill Osborne, Stephen Tang, Jingyuan Chen,
F. Donelson Smith, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill
Email: {jbakita, shareef, shosborn, sytang, leochanj, smithfd, anderson}@cs.unc.edu

***Abstract*—Simultaneous multithreading (SMT) enables enhanced computing capacity by allowing multiple tasks to execute concurrently on the same computing core. Despite its benefits, its use has been largely eschewed in work on real-time systems due to concerns that tasks running on the same core may adversely interfere with each other. In this paper, the safety of using SMT in a mixed-criticality multicore context is considered in detail. To this end, a prior open-source framework called MC^2 (mixed-criticality on multicore), which provides features for mitigating cache and memory interference, was re-implemented to support SMT on an SMT-capable multicore platform. The creation of this new, configurable MC^2 variant entailed producing the first operating-system implementations of several recently proposed real-time SMT schedulers and tying them together within a mixed-criticality context. These schedulers introduce new spatial-isolation challenges, which required introducing isolation at both the L2 and L3 cache levels. The efficacy of the resulting MC^2 variant is demonstrated via three experimental efforts. The first involved obtaining execution data using a wide range of benchmark suites, including TACLeBench, DIS, SD-VBS, and synthetic microbenchmarks. The second involved conducting a large-scale overhead-aware schedulability study, parameterized by the collected benchmark data, to elucidate schedulability tradeoffs. The third involved experiments involving case-study task systems. In the schedulability study, the use of SMT proved capable of increasing platform capacity by an average factor of 1.32. In the case-study experiments, deadline misses of highly critical tasks were never observed.**

I. Introduction

In many safety-critical application domains, a trend is underway to shift from unicore platforms to multicore ones. An attractive attribute of multicore platforms in these domains is the ability to support computationally intensive workloads within a restricted size, weight, and power (SWaP) envelope. However, this attribute may come at the price of added monetary cost. Thus, when multicore platforms are employed, any waste of processing capacity should be avoided: excessive waste can necessitate added hardware, increasing cost.

When examining multicore platform choices today, we find many offerings that provide multiple hardware threads per core that can execute tasks concurrently. This *simultaneous multithreading (SMT)* can enhance computing capacity, but has been largely eschewed in work on real-time systems due to concerns that tasks running on the same core may adversely interfere with each other in contending for CPU resources. Our recent work has called this point of view into question by noting that multicore platforms are rife with other interference sources, like caches, buses, and memory banks, that are not seen as fundamental impediments [1]–[3]. Indeed, as discussed below, extensive recent work has shown that these other interference sources *can* be safely mitigated.

We examine herein whether the potential ill effects of SMT can also be safely mitigated, enabling its benefits to be reaped in safety-critical embedded real-time systems. In such systems, tasks are often partitioned among different *criticality levels* and we assume that here. While it may seem self-evident that SMT can be safely used for tasks that are not very critical (and thus more interference-tolerant), *the range of criticality levels across which SMT can be safely applied is not clear.*

In this paper, we present the results of a research study undertaken to clarify this issue. This study points to the conclusion that SMT *is* safe to apply to increase platform capacity, even for tasks of high criticality. We elaborate on this conclusion below, after first providing an overview of prior related work to provide context.

Related work. When moving from unicore to multicore platforms in supporting safety-critical applications, the main new complication that arises is dealing with hardware components such as caches, memory, buses, etc., that can be sources of interference when tasks on different cores contend for them. For a given interference source, two basic options arise: interference due to that source can be eliminated altogether (e.g., by using cache-partitioning techniques for shared caches), or its ill effects can be carefully accounted for when determining task execution-time bounds. Many nuances arise in applying these basic options, and the literature exploring these nuances is now quite extensive [4]–[46].

Historically, the real-time community has paid relatively little attention to the potential of SMT. However, users in industry are eager to make more use of SMT; in particular, multiple developers have expressed interest to the U.S. Federal Aviation Administration (FAA) in using SMT in safety-critical systems [47]. Motivated by this industrial interest, we have considered the application of SMT to real-time systems in three recent papers. In the first of these papers [2], the effects of SMT on worst-case execution times (WCETs) is investigated, and it is demonstrated that in a soft real-time

Work was supported by NSF grants CNS 1563845, CNS 1717589, CPS 1837337, CPS 2038855, and CPS 2038960, ARO grant W911NF-20-1-0237, and ONR grant N00014-20-1-2698.

context, SMT can improve schedulability by 30% or more. In the second paper [1], a static (i.e., table-based) scheduling approach is presented that enables SMT to be used in hard real-time systems. Devising this approach entailed developing a new form of measurement-based probabilistic timing analysis (MBPTA) to obtain safe upper bounds on execution times when SMT is in use.[1] In the third paper [3], a priority-based scheduling method is given for hard real-time systems using SMT that can be more flexibly applied.

One limitation of these papers is that SMT is the only potential interference source that is considered. As noted above, many other interference sources exist on a multicore platform. Determining how to properly mitigate all such sources holistically requires sifting through many tradeoffs.

Contributions. In this paper, we extend prior work on the usage of SMT in real-time systems by considering systems with multiple criticality levels and interference sources. We do this by incorporating support for SMT into a prior open-source framework called MC^2 (mixed-criticality on multicore) [8], [9], [18], [19], [24], [27]–[29] that provides criticality-aware isolation mechanisms for mitigating cache and DRAM interference. Our contributions into the effectiveness of SMT in mixed-criticality systems can be divided into four parts.

First, we extended the existing MC^2 framework by porting it from the relatively simple quad-core ARM platform assumed in prior work to a significantly more capable 16-core AMD platform. While the prior ARM platform provides no SMT support and only two cache levels, the AMD platform allows SMT and has three levels of cache, allowing for more nuanced scheduling decisions. Using this new platform, we implemented a novel multi-level cache isolation mechanism for *both* the L2 and L3 caches based on Linux's Non-Uniform Memory Access (NUMA) system that subsumes and improves upon prior isolation techniques. To our knowledge, this is the first work to consider multi-level cache management.

Second, we modified MC^2 to enable the usage of SMT across all criticality levels by leveraging previously proposed real-time SMT schedulers [1]–[3]. While prior work on these schedulers focuses on their theoretical properties, we implemented an MC^2 scheduler that subsumes these preexisting SMT scheduling approaches. To our knowledge, we are the first to implement a scheduler that supports SMT in a mixed-criticality system with hardware-isolation features.

Third, to assess how SMT affects task execution times under various cache allocations, we undertook extensive benchmarking efforts, the results of which are discussed herein.

Fourth, to understand how SMT impacts schedulability, we conducted both a large-scale overhead-aware schedulability study and case-study experiments. In the former, parameter choices were informed by our benchmark data; in the latter, such choices were further constrained to align with systems deemed "hard to schedule" in the schedulability study. In the schedulability study, we found that, compared to using cache-management techniques alone, allowing SMT increased schedulable utilization by a factor of 1.32 on average and over 1.46 in most cases. In the case-study experiments, no deadline misses of highly critical tasks were ever observed.

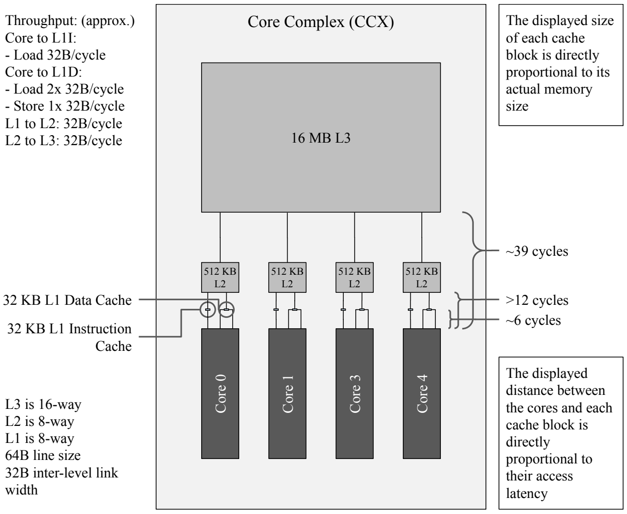

Fig. 1. Diagram of a single core complex on the AMD Ryzen 3950X. Note that the spacing of caches in this figure is proportional to their documented latencies and their displayed sizes are proportional to their data capacity.

Organization. The rest of this paper is organized as follows. After providing necessary background information in Sec. II, we describe our new MC^2 implementation in Sec. III. In Sec. IV, we discuss the modifications we made to MC^2's scheduling policies to enable SMT. We then present the results of our benchmark experiments, our overhead-aware schedulability study, and our case-study experiments in Secs. V–VII, respectively. Finally, we conclude in Sec. VIII.

II. Background

In this section, we describe our task model, hardware platform, SMT, and mixed-criticality scheduling in MC^2.

A. Task Model

We consider the implicit-deadline sporadic/periodic task model and assume familiarity with this model, though it will be extended in later subsections to consider mixed criticality and SMT. We consider a task system of n tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$. The a^{th} job of task τ_i is denoted $\tau_{i,a}$. The period and execution cost of τ_i are denoted T_i and C_i, respectively. The tardiness of a job $\tau_{i,a}$ is $\tau_{i,a}$'s completion time subtracted by its deadline, or 0 if $\tau_{i,a}$ completes prior to its deadline. The tardiness of task τ_i is then the supremum of the tardiness of its jobs. A task is *hard real-time (HRT)* if it requires zero tardiness. Alternatively, a task is *soft real-time (SRT)* if it only requires bounded tardiness.

B. Hardware Platform

In this work, we ported MC^2 from the previously considered quad-core ARM machine to a more sophisticated x86-based platform, the AMD Ryzen 3950X, depicted in Fig. 1. This new platform choice enables features to be examined such as

[1] MBPTA is a family of timing-analysis methods that give probabilistic upper bounds on WCETs based on sample observed execution times.

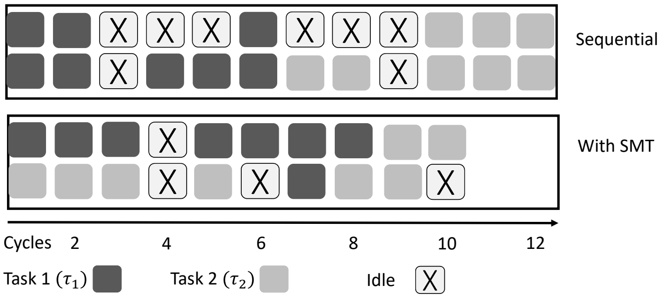

Fig. 2. Execution of tasks τ_1 and τ_2 sequentially and with SMT.

higher core counts, SMT, and larger caches shared at multiple levels. All of these features have been announced in ARM's next generation of embedded processors [48].

The 3950X features 16 cores with two hardware threads each. The cores are equally divided among four core complexes (CCXs). Each CCX has a 16MB L3 cache that is shared among all cores in the complex. Hardware support for partitioning L3 caches and memory buses is included. Each core has a 512KB L2 cache and separate 32KB L1 caches for instructions and data. L2 and L1 caches are shared between threads. Caches are coherent across CCXs, but one CCX cannot initiate cache fills to another CCX. While the previous MC^2 provides DRAM bank isolation, we do not consider that feature in this paper,[2] so we refrain from commenting on DRAM-related details concerning the 3950X here.

C. SMT Overview

Modern computing cores use instruction-level parallelism within jobs to execute multiple instructions per cycle. Enabling SMT takes this behavior further by allowing multiple jobs to execute instructions within a single cycle. An overview is given in Fig. 2 and Ex. 1 below, both of which closely follow explanations in [1], [3]. Further information on the fundamentals of SMT can be found in the work of Eggers et al. [49]. For a detailed discussion of factors that can affect SMT execution in practice, see [50], [51].

Example 1. *At the top of Fig. 2, jobs of tasks τ_1 (darker) and τ_2 (lighter) execute sequentially without SMT on a core that can accept two instructions per cycle. When fewer than two instructions are ready, as in cycles 3 through 5, cycles are wasted. τ_1 finishes at the end of 6 cycles and τ_2 at the end of 12. In the bottom half of the figure, the same jobs employ SMT to execute in parallel. This reduces the number of lost cycles. τ_1 finishes after 8 cycles and τ_2 after 10. SMT thus delays the completion of τ_1, but speeds up the completion of τ_2 since it must not wait for τ_1 before beginning its own execution.* ◀

SMT's potential is obvious: more work can be done in less time. For real-time systems, the downside is two-fold. First, the presence of an additional job on the same core introduces new sources of interference that, if not carefully accounted for, could lead to inaccurate task execution costs, and hence, unsafe systems. Second, individual job execution costs will generally increase, requiring careful balancing of the tradeoffs between increased throughput and higher individual costs.

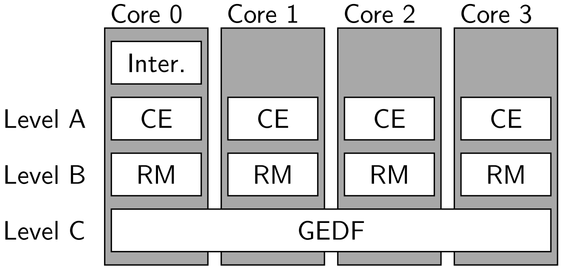

Fig. 3. Scheduling under preexisting MC^2.

We manage the first problem through a novel cache partitioning techniques to isolate SMT-using jobs from one another, described in Sec. III. We address the second problem via new scheduling policies described in Sec. IV. It is true that we cannot perfectly isolate SMT-enabled jobs, but, as already mentioned in Sec. I, any lack of isolation can be mitigated by accounting for it, and this is the case with or without SMT.

D. Preexisting MC² Framework

The MC^2 scheduler is implemented as a plugin to the LITMUS^{RT} kernel [52]–[54], a patch to Linux that eases the process of implementing real-time schedulers for research purposes. MC^2 employs a mixed-criticality task model following Vestal [55], with three criticality levels, A, B, and C. (There is an additional Level D for best-effort workloads, but we omit it here as it provides no real-time guarantees.) Level A consists of tasks where the consequences of failure would be catastrophic, Level B consists of tasks where any failure is hazardous, and Level C consists of tasks where failure is less severe. As such, Level-A and -B tasks are HRT while Level-C tasks are SRT. We use n_L to denote the number of tasks at Level L (A, B, or C).

Scheduling in MC^2. Fig. 3 depicts the preexisting hierarchical MC^2 scheduler on its original four-core platform. This scheduler statically prioritizes higher-criticality levels over lower ones. Each level employs its own internal scheduler depending on the real-time requirements at that level. Level-A and -B tasks, which are HRT, are respectively scheduled via a table-driven cyclic executive (CE) and rate monotonic (RM) scheduler. Such tasks are partitioned onto cores. Level-C tasks, which are SRT, are scheduled via a global earliest-deadline-first (GEDF) scheduler. Interrupts are redirected to Core 0 and are serviced at a higher priority than all tasks on this core.

Period restrictions. To simplify schedulability analysis, it is assumed in prior MC^2 works that Level-A and -B tasks are released synchronously and periodically and that all periods of such tasks on each core are harmonic. Also, it is assumed that the hyperperiod of all Level-A tasks on any core evenly divides the period of any Level-B task on that core. These assumptions simplify the schedulability analysis of Level-B tasks, which are scheduled by RM. Such assumptions are

[2] In prior work on MC^2, cache management was seen to have a much greater impact than providing DRAM bank isolation. Moreover, when introducing SMT, cache-related issues are much more of a concern.

unneeded for Level-C tasks, which are statically prioritized below Level-B tasks. As such, Level-C tasks may be sporadic and need not be harmonic with respect to other tasks.

Mixed-criticality analysis. In line with the seminal work by Vestal [55], each task is assigned a distinct *provisioned execution time (PET)* at each of Levels A, B, and C. A PET represents an upper bound on the execution cost of a task under a level of pessimism that increases with higher criticality levels. This reflects that the consequences of failure depend on the criticality of the failing task. In prior MC^2 work, Level-A, -B, and -C PETs are, respectively, inflated worst-case, worst-case, and average-case execution times. We use PET_i^L to denote the Level-L PET of task τ_i, and PET_i to denote τ_i's PET at its own criticality level. For each PET, a task has a corresponding *utilization*, $u_i^L = \frac{\text{PET}_i^L}{T_i}$. References to different per-level PETs below also imply different utilizations.

Schedulability is determined for each criticality level. At Level A, it is sufficient that Level-A tasks meet their required real-time guarantees so long as no job of a Level-A task exceeds its Level-A PET. At Level B, it is sufficient that Level-A and -B tasks meet their guarantees when no Level-A or -B jobs violate their Level-B PETs. Finally, at Level C, all tasks must meet their guarantees under Level-C PETs. If the system is schedulable for each criticality level, then the system as a whole is schedulable.

Cache partitioning. In the preexisting MC^2 framework, cross-core interference caused by concurrently executing tasks is reduced by providing *spatial isolation* to high-criticality tasks. This often reduces to partitioning shared hardware resources, with one of the most thoroughly considered resources being shared caches. The LLC (last-level cache) in prior MC^2 work, and our work, is set-associative, meaning there is a many-to-one mapping of physical addresses to each cache set of 16 cache lines. By managing the physical addresses used by any task ("coloring"), we can control the cache sets it uses. Additionally, we can control the specific cache lines ("ways") each core uses inside any set via a per-core mask.

The old MC^2 platform allowed for 16 cache colors [56] and 16 cache ways. Fig. 4 depicts the default cache partitioning considered in most prior work on MC^2. As shown, Level-A/B tasks on each core receive disjoint cache colors, and disjoint ways relative to Level-C tasks and the operating system (OS). This prevents Level-A/B tasks on one core from evicting any cache lines of Level-A/B tasks on other cores. It also prevents Level-C tasks or the OS from evicting Level-A/B cache lines.

As reducing the amount of cache available to a task will increase its execution time, the partition sizes in Fig. 4 have a significant impact on system schedulability. As such, a major component of prior MC^2 frameworks is a mixed-integer linear program (MILP) that resizes these partitions (by adjusting the dashed lines) to find a partitioning of the cache that results in a schedulable system. Due to constraints detailed in Sec. III-A, we only consider allocating LLC space by way, rather than using a scheme like that in Fig. 4.

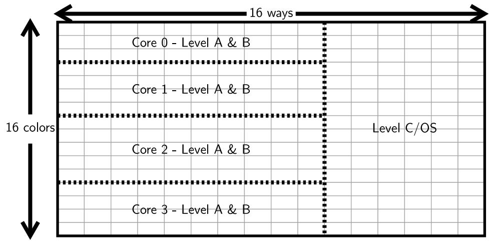

Fig. 4. Example LLC cache partitioning from prior MC^2 work.

Adding SMT to MC^2. Allowing SMT requires altering our task model. For Levels A and B, we allow SMT only via the *simultaneous co-scheduling* of all jobs of a given pair of tasks. (Recall that the 3950X has two threads per core.)

Def. 1. *[1] Two jobs are* simultaneously co-scheduled *if both begin execution simultaneously on the same core, and after one job completes, no other job executes simultaneously with the remaining job. We let $\tau_{i.a:j.b}$ denote the simultaneously co-scheduled jobs $\tau_{i.a}$ and $\tau_{j.b}$.* ◄

Simultaneously co-scheduled jobs require their own definitions for PETs.

Def. 2. *[1] The* joint PET *to simultaneously execute jobs of τ_i and τ_j, denoted by $PET_{i:j}$, is defined as the execution time for both jobs assuming they begin simultaneously. In Fig. 2, the joint PET of τ_1 and τ_2 is given by $PET_{1:2} = 10$. If $i = j$, then $PET_{i:j} = PET_i$, indicating solo execution for τ_i.* ◄

For Level-C tasks, where less precise PETs may be acceptable, we simply give each task an additional PET to account for SMT being in use, without considering what other task may be sharing the same core. We denote the PET of a Level-C task τ_i by PET_i^* when SMT is in use. Note that, when referring to Level-C tasks, there is never a need to refer to Level-A or -B PETs. Also, all PETs, regardless of the assumed criticality level, will depend on how cache space is allocated.

E. Determining PETs

To determine per-criticality-level PETs, we follow the same measurement-based approach used in prior work on MC^2 [8], [9], [18], [19], [24], [27]–[29]. Specifically, we determine a task's Level-B (resp., Level-C) PET by measuring its worst-case (resp., average-case) execution cost. We determine Level-A PETs by applying a 50% inflation factor to Level-B PETs. Such inflation is consistent with industry practice as reported by Vestal [55]. We derive joint PETs using methods given in [1], which are discussed in Sec. V-B in detail.

III. Implementing Isolation with Threading

To use our new hardware platform, we ported the existing open-source ARM-based MC^2 framework to our x86 3950X platform. This effort required addressing two challenges. First, to adapt and extend the spatial isolation in the preexisting MC^2 to match our platform's capabalities, we had to modify MC^2 to partition both our platform's per-core L2 and per-CCX L3

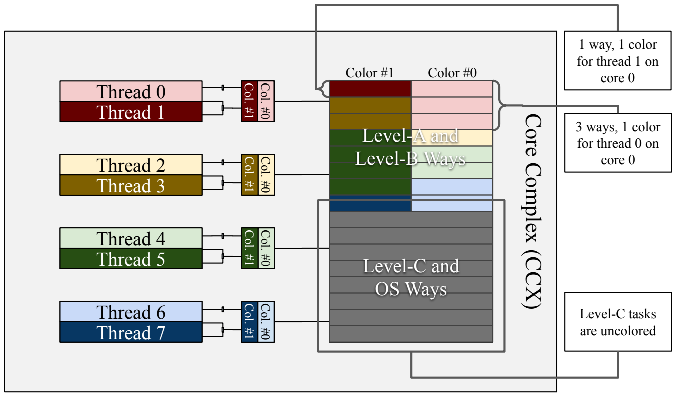

Fig. 5. Example of an asymmetrical cache allocation on a CCX in our system.

caches by ways and sets. We did this using a novel NUMA-based approach that subsumes and improves prior isolation techniques. Second, we needed to use our cache-partitioning techniques to provide isolation between threads. Providing this isolation simultaneously within multiple levels of cache introduces complexities not considered in the preexisting MC^2. In total, our porting effort involved modifying or adding 3,000 lines of code in MC^2, as well as reverse-engineering efforts to deduce how caches are mapped on our platform. In the following subsections, we describe how we partition multiple levels of caches and enable isolation between threads.

A. Partitioning the L3

As shown in Fig. 5, we can partition the L3 cache on our platform via sets and ways, like the shared L2 cache in the preexisting MC^2. Way-based partitioning is straightforward, as we can use AMD's QoS extensions to allocate ways analogously to the lockdown registers used previously [57]. Cache coloring is also aided by the fact that the 3950X linearly maps addresses to cache sets. We confirmed that our platform employs such a linear mapping via micro-benchmarks.

Our cache-coloring implementation provides colors by modifying Linux's NUMA system, which is typically used when physical memory is divided among different nodes with different access times. This differs from the prior MC^2 and other works that consider cache coloring under Linux, which typically implement colors by modifying Linux's buddy allocator. At a high level, both approaches implement colors by dividing physical pages into groups such that the cache sets mapped from pages in one group are disjoint from sets mapped from other groups. Our coloring implementation is simpler because the NUMA system contains preexisting functionality for dividing memory using logical nodes.

NUMA-based cache coloring. Linux typically organizes physical memory into a hierarchical structure of *nodes* and *zones*. Nodes represent independent physical memory controllers on NUMA machines. However, the Linux kernel is not aware of subtleties like NUMA until it is well into initialization. To address this, the kernel uses the boot-time *memblock* allocator until it has parsed the NUMA configuration into memory nodes and zones. Linux retires this allocator by freeing unused memblocks to the buddy allocator, setting up its stack with pages from the buddy allocator, jumping to that new stack, and then freeing all the remaining data in the memblock allocator to the buddy allocator. Note that this process sends every page in the system through the buddy allocator page free function before it gets used permanently.

Linux also can "fake" a NUMA system of n nodes, even when the system has one memory controller, by splitting physical memory into n blocks and assigning one block to each fake NUMA node. From userspace and most of the kernel, these nodes are indistinguishable from the real ones.

By creating fake NUMA nodes for each of our memory colors and modifying the aforementioned page free function to be color-aware (make it return pages not to the requested node, but to the node that we know matches its color), we implement a remarkably simple yet completely holistic coloring implementation. This allows kernel, driver, and task data to all be easily moved between or interleaved across colors simply by using the existing NUMA management facilities.

Our new implementation has several benefits. First, we only needed to modify **23 lines** of code to implement coloring. Second, a simpler implementation has also reduced the likelihood of bugs.[3] Finally, as coloring is implemented via the NUMA system, the interface to the NUMA system provides a convenient and efficient method for configuring coloring. This advantage is desirable for systems that require isolation while interacting with memory dynamically, which we hope to explore in future work. Other works [58], [59] have also used fake NUMA nodes; however, not for cache coloring.

B. SMT and Hierarchical Caches

Per-core caches were not a concern in the old MC^2, as a core only permitted a single thread of execution. With SMT on our platform, threads on the same core may evict each other's cache lines if not isolated from one another within the per-core L2 caches. Thus, we must provide isolation within the per-core L2 caches and the per-CCX L3 caches simultaneously.

We could partition L3 caches on our platform using similar mechanisms to the LLC cache of the old MC^2, however, this would inhibit per-core L2 partitioning. Our L2 cache has no facility for way-based partitioning, so it must be colored—an operation that also maps into the L3 cache. Thus, any coloring decision in the L2 cache will also apply to the L3 cache. This reduces the number of potential cache partitions that can be considered on our platform.

The number of potential partitions is further reduced when Linux requests ranges of physically contiguous memory, such as in drivers for DMA buffers. As the maximum size of contiguous memory that can be allocated is inversely proportional to the number of cache colors supported, we must limit this number. Fortunately, we can provide isolation in the L2 caches to threads so long as we support at least two colors, as shown in Fig. 5. We discuss how our L2 and L3 cache-allocation approaches are integrated holistically in Sec. IV-C.

[3] While porting to the 3950X, we found significant edge-case bugs in the memory- and cache-management code of the prior MC^2 implementation.

While we do not provide L1 isolation for a core's two threads, we account for any L1 interference in our measurement process for determining PETs, as discussed in Sec. V. In general, we address SMT inter-thread interference sources other than L2 caches by either applying temporal partitioning, thus preventing threads from executing concurrently, or by accounting for this interference in our measurement process. We briefly consider other interference sources next.

C. Other Sources of Interference

Such sources include DRAM banks and buses, cache-coherence traffic, and peripheral interrupts on our platform.

We account for DRAM bank and bus interference by measuring PETs in presence of contending tasks running on all other CCXs in the system. The only sources of interference between different CCXs are cache-coherency traffic and DRAM traffic. We prohibit tasks from sharing memory between CCXs and account for OS cache-coherency traffic separately, as in prior MC^2 work. We redirect other OS activities to Core 0, on which we do not schedule any real-time task. In a real system, we can account for the delays caused by cache-related OS interference via overhead-accounting techniques by Brandenburg [54, pages 230–235]. While we deal with DRAM- and bus-related interference effects by accounting for them, an alternative is to introduce isolation mechanisms for these interference sources as well. In this first work on using SMT in a mixed-criticality context providing isolation, we chose not to do this, though we may do so in future work.

We handle interrupts that are not required to schedule tasks by redirecting all of them to Core 0, which is dedicated to interrupt processing (Fig. 6).

IV. Scheduling and Partitioning with SMT

In this section, we describe modifications we made to MC^2's scheduler to enable SMT on our new hardware platform. This entailed addressing three key challenges. First, we needed to ensure predictable co-scheduling of Level-A and -B jobs per Def. 1. Second, we wanted to enable tasks that can benefit from SMT to use it, while still allowing other tasks to execute without SMT. Third, our AMD platform has many more cores than the prior ARM platform, which motivated us to add support for clustered (rather than global) scheduling at Level C.

Modified MC^2 scheduler. Our modified hierarchical MC^2 scheduler for two CCXs is depicted in Fig. 6(c). The remaining CCXs are omitted due to redundancy. While the preexisting MC^2 schedules Level-C tasks by GEDF, we schedule them by clustered EDF (CEDF). We discuss this in Sec. IV-B in detail. Note from Fig. 6(c) that we have chosen not to schedule real-time tasks on Core 0, which is responsible for processing interrupts. This differs from the preexisting MC^2. We have chosen this because scheduling on this core introduces overhead-accounting complexities that are orthogonal to the issue of whether SMT improves schedulability under MC^2. Also, the loss of a single core's capacity is less substantial on our 16-core platform than the previous quad-core platform.

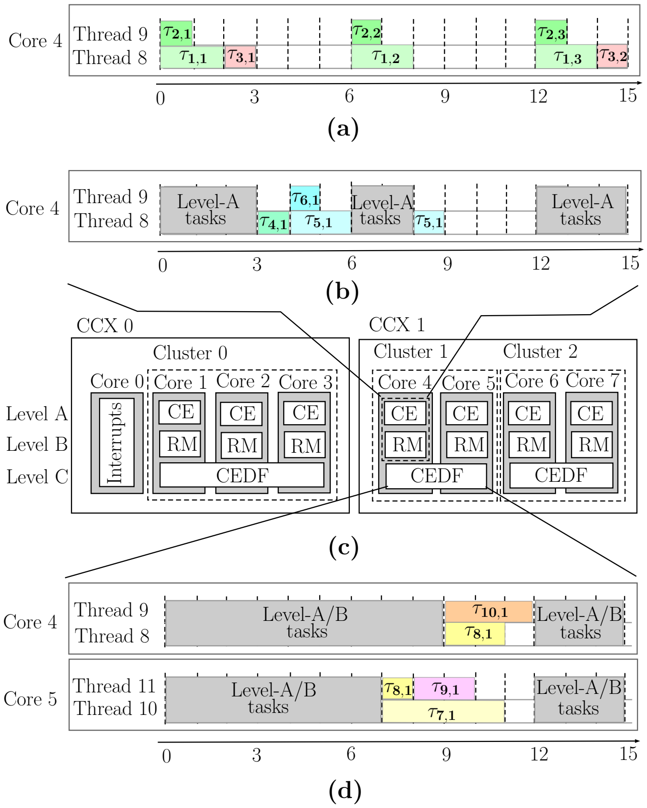

Fig. 6. Schedule of (a) Level-A tasks, (b) Level-B tasks in Ex. 2, (c) MC^2 scheduling on our new platform, and (d) schedule of Level-C tasks in Ex. 2.

Note that timer interrupts and inter-processor interrupts (IPIs) needed to trigger the scheduler occur on each core, and these are included in our overhead accounting.

Resource allocation under prior MC^2 frameworks has included a task-to-core partitioning step (see Levels A and B in Fig. 3) followed by a cache-partitioning step (see Fig. 4). Our modified framework follows this paradigm, though we made changes due to SMT and the grouping of cores into CCXs. We begin by describing our modified partitioning step.

A. Partitioning Level-A and -B Tasks

To make use of threading for Level-A tasks, we must reconcile the potential interference effects caused by threading with MC^2's goal of reducing interference for high-criticality tasks. Following our previous work [3], we accomplish this by requiring that any Level-A task is simultaneously co-scheduled (Def. 1) with the jobs of at most one other Level-A task with the same period.[4] Thus, any threaded Level-A task is only ever interfered with via SMT by its co-scheduled task, making it much simpler to consider such tasks under Level-A pessimism. Extending the terminology used for jobs in Defs. 1 and 2, we refer to such co-scheduled tasks as "paired tasks" and to any unpaired Level-A tasks as "solo tasks." We denote two paired tasks τ_i and τ_j by $\tau_{i:j}$. Note that, as our co-scheduling restrictions prevent an idle thread from executing any workload while its corresponding other thread executes a Level-A task,

[4] In many safety-critical domains, only a few period choices are used.

from an analysis point of view, two paired tasks are similar to a logical solo task with period equal to the pair's shared period and execution cost equal to the joint cost of the paired tasks. Thus, we can partition the paired tasks identically to solo tasks. We partition Level-A tasks by worst-fit and schedule them by a CE scheduler, as in prior MC^2 work.

This leaves the question of how to decide which tasks are paired. We make this decision using an integer linear program (ILP) that seeks to minimize total utilization while not creating paired tasks with excessively high utilization. This ILP is virtually identical to one we presented previously in [3]. The primary difference is that, while we required in [3] only that paired tasks have utilizations not exceeding 1.0, we now consider this utilization cap as a tunable parameter in our framework. In preliminary experiments, we compared using 1.0, 0.75, and 0.5 for this value and found that 0.5 generally gave the best results. The experiments discussed in Sec. VI all use 0.5. Following [1], we also do not consider pairing tasks with PETs differing by 10× or more. When making these initial task-pairing decisions, we use task utilizations with respect to Level-A PETs under full cache allocations. This is necessary because cache partition sizes are undecided by this stage of the framework.

Example 2. *Consider three Level-A tasks τ_1, τ_2, and τ_3 with periods 6, 6, and 12, respectively. Assume that $\tau_{1:2}$ and τ_3 become a paired task and a solo task, respectively, and are assigned to Core 4. Fig. 6(a) depicts a possible CE schedule of $\tau_{1:2}$ and τ_3 on Core 4 up to time 15. Thread 9 is idle during the intervals $[1, 2)$ and $[2, 3)$ due to the co-scheduling restriction of paired tasks and τ_3 being a solo task, respectively.* ◀

Similar to Level-A tasks, we require each Level-B task to be simultaneously co-scheduled with at most one other Level-B task with the same period. We decide the Level-B tasks pairings using the same ILP as for Level A. We partition Level-B tasks after partitioning Level-A tasks using the same methodology used for partitioning Level-A tasks with two differences. First, we consider task utilizations with respect to Level-B PETs. Second, we reduce the initial capacity of each core by the total utilization of the Level-A tasks already partitioned on that core.

Example 2 (cont'd)**.** *Suppose Level-B tasks τ_4 and $\tau_{5:6}$ with periods 12 and 24, respectively, are assigned to Core 4. Fig. 6(b) shows an RM schedule for these tasks.* ◀

B. Scheduling Level-C Tasks

As Level C is of lower criticality than Levels A and B, we do not impose co-scheduling restrictions on Level-C tasks that execute as threads. Such tasks are scheduled via EDF and allowed to migrate, as in prior MC^2 work. Level-C tasks in prior MC^2 work were scheduled globally across all cores, and thus, did not require any partitioning. This is undesirable under our new platform for two reasons. First, we wish to restrict migrations of Level-C tasks such that they remain within a CCX, thereby allowing them to maintain L3 cache locality. Second, we would like to support Level-C tasks that execute both alone (i.e., solo tasks) and as threads (i.e., threaded tasks). If we required all Level-C tasks to be threaded, per-task utilizations may increase to the point that the system is unschedulable. As schedulability analysis does not exist for situations where both threaded and solo tasks contend for the same cores, the set of solo Level-C tasks must be partitioned away from the set of threaded Level-C tasks. A similar approach was used in our prior work [2].

We designate a Level-C task as threaded if doing so is expected to reduce total system utilization, so long as its threaded task utilization does not exceed a tunable cap. Otherwise, we designate the task as a solo task.

After deciding which Level-C tasks are threaded, we partition the set of cores into clusters (see Fig. 6(c)) and schedule Level-C tasks on these clusters via CEDF. We require that each such cluster contains either only solo tasks or only threaded tasks (thereby separating these two task types) and that clusters do not cross CCX boundaries (thereby preventing Level-C tasks from migrating outside of a CCX). We refer to clusters that contain only solo (resp., threaded) tasks as *solo* (resp., *threaded*) *clusters*. We partition the cores into clusters by first designating each core as either solo or threaded in proportion to the total utilization of tasks per category, and then grouping cores into clusters so that at most one CCX contains both a solo and a threaded cluster. We assign tasks to clusters using a heuristic loosely based on the worst-fit decreasing heuristic, in which we consider the solo (resp., threaded) tasks in decreasing order of utilization and assign each task to the solo (resp., threaded) cluster with the maximum possible remaining capacity while maintaining Level-C schedulability.

Example 2 (cont'd)**.** *Suppose Level-C tasks τ_7, τ_8, τ_9, and τ_{10} are designated as threaded tasks and are assigned to (the threaded) Cluster 1 in Fig. 6(c). A CEDF schedule of these tasks is shown in Fig. 6(d). Job $\tau_{8,1}$ is preempted by the higher-priority job $\tau_{9,1}$ on Core 5 and later resumes on Core 4. Note that there is no co-scheduling restriction here: both $\tau_{8,1}$ and $\tau_{9,1}$ execute on Thread 11 while $\tau_{7,1}$ executes on Thread 10. Note that if Cluster 2 is designated as a solo cluster, then scheduling within it would be similar, except that only one Level-C task would execute on a given core at a time.*

C. Cache Allocation and Overheads

After making all threading-, core-, and cluster-assignment decisions, we allocate ways of each CCX's L3 cache on a per-core basis for Level-A/B tasks and a per-CCX basis for Level-C tasks. Each of these ways allocated to a core running Level-A/B tasks is further subdivided between the two threads on that core by cache coloring. As discussed in Sec. III-B, this partitions both threads from each other in the core-local L2 cache. An example cache partitioning is depicted in Fig. 5.

We determine cache partition sizes using a prior MC^2 MILP by Chisholm et al. [9]. The decision variables of this MILP are the number of cache ways to allocate to each core, and the Level-C partition. The objective of this MILP is to minimize the total Level-C utilization of the task system while satisfying the schedulability conditions of all criticality levels; these conditions are covered in Sec. IV-D. In applying these conditions, we account for the following overhead sources: cache-related delays due to preemptions and migrations, job release costs and latencies, IPIs, scheduling costs, and context-switch costs. We account for cache-related delays and overheads due to interactions between different criticality levels as done in [19], and use techniques from Brandenburg [54][5] for all other overheads. In addition to the overheads mentioned above, paired Level-A and -B tasks incur an additional IPI as they dispatch two, rather than one, tasks at a time. We include this IPI in our overhead accounting.

The L3 cache-isolation techniques described in Sec. III could potentially be used to allocate an asymmetric number of ways to two threads on the same core without compromising any isolation guarantees. This is the case in Fig. 5, where Thread 0 is allocated three ways while Thread 1 is allocated one, even though both threads are located on the same core. The cache partitions possible with asymmetric cache allocation subsume the partition options considered by the MILP in the prior paragraph; however, asymmetric allocations break fundamental assumptions of our optimization framework, resulting in potentially safe cache allocations being deemed unschedulable. As such, we do not consider asymmetric cache allocations and defer exploration of this issue to future work.

D. Schedulability under MC² with SMT

Schedulability analysis under MC^2 at Levels A and B is largely unchanged from that of [9] by introducing SMT; due to the nature of simultaneous co-scheduling, any paired task is identical to a solo task from a scheduling perspective.

Schedulability analysis at Level C is altered. In order to apply preexisting MC^2 schedulability analysis to a threaded Level-C cluster, each thread must be viewed as its own "core" from the perspective of the analysis. Consider a cluster of m cores. Let $U^C_{A,p}$ (resp., $U^C_{B,p}$) denote the total Level-C utilization of all paired and solo tasks on core p at Level-A (resp., -B). Let h, H_k, and U^C_C denote the largest, the sum of the $\min(k, n_C)$ largest, and the sum of all utilizations of Level-C tasks. From [60], without Level-C threading, the Level-C schedulability condition for tasks in this cluster is

$$U^C_C + \sum_{p=1}^{m} \left(U^C_{A,p} + U^C_{B,p}\right) \leq m$$

$$(m-1)h + H_{m-1} + \sum_{p=1}^{m} \left(U^C_{A,p} + U^C_{B,p}\right) < m.$$

With Level-C threading, we effectively double the core count from the perspective of the analysis, so this becomes

$$U^C_C + 2\sum_{p=1}^{m} \left(U^C_{A,p} + U^C_{B,p}\right) \leq 2m$$

$$(2m-1)h + H_{2m-1} + 2\sum_{p=1}^{m} \left(U^C_{A,p} + U^C_{B,p}\right) < 2m.$$

V. Benchmark Experiments

In this section, we report on the results of benchmark experiments that show how tasks running with SMT are influenced by cache effects and how PETs differ across criticality levels. Results from these experiments are used to inform the choices of variables used in our schedulability study. The code and instructions for all our experiments is available online.[6]

A. Benchmark Selection

In performing our benchmark experiments, we utilized the TACLeBench sequential benchmarks [61] ("TACLe"), the DIS Stressmark suite [62] ("DIS“), and the San Diego Vision Benchmark suite [63] ("SD-VBS“). The TACLe benchmarks emulate tasks common to embedded systems. Their runtimes and working-set sizes (WSSs) are small. The DIS benchmarks perform data-intensive functions such as matrix multiplication or image processing with configurable runtimes and WSSs. The SD-VBS benchmarks consist of both computational and data-intensive programs that emulate computer-vision applications. Their runtimes and WSSs vary depending on input size. We explored the behavior of SMT in MC^2 across a wide variety of cache allocations and memory usages, generating over 200GB of timing data.[7]

All results use sample sizes of at least 1,000 and, where configurable, use a WSS of 2MB unless otherwise noted. All experiments were run concurrently with contending synthetic tasks using preexisting techniques [19] to continually generate read and write requests to DRAM. Tasks were also executed in an environment designed to minimize scheduling noise.[8]

B. SMT and Multi-Level Cache Isolation

As described earlier, the L2 cache on our test platform is shared by two threads on the same core. This introduces a source of interference that can have a significant impact on the execution times of both threads. Further, the threads executing on all four cores of a CCX share the same L3 cache, another source of interference. In this subsection, we examine how cache isolation affects the benefits of SMT.

We previously gave a metric, $M_{i:j}$, to quantify the benefit of SMT [1]. Here, we apply $M_{i:j}$ to Levels A and B. Under a given cache allocation scheme, $M_{i:j}$ satisfies $\text{PET}_{i:j} = \text{PET}_i + M_{i:j} \cdot \text{PET}_j$, or $M_{i:j} = \frac{\text{PET}_{i:j} - \text{PET}_i}{\text{PET}_j}$. If $M_{i:j} < 1$, then pairing τ_i and τ_j under that cache allocation scheme may be beneficial as their joint cost is less than the sum of their solo costs. If $M_{i:j} \geq 1$, then pairing them is not beneficial. Note that, smaller $M_{i:j}$ values indicate larger SMT benefits.

Recall from Sec. II that Level-C tasks do not employ simultaneous co-scheduling. Thus, to quantify SMT at Level C, we

[5] For overhead-accounting details, see [54, pages 219–259]; for a summary, see [54, pages 261–263].

[6] See the artifact link on https://www.cs.unc.edu/~anderson/papers.html.

[7] See our online appendix for full benchmark information and tables of supplemental results: https://www.cs.unc.edu/~anderson/papers.html

[8] Specifically, we use `sched_yield()` and `wbinvd` between samples and Linux's `nohz_full`, `isolcpus`, and `irqaffinity` cmdline params.

TABLE I
SMT Effectiveness at Levels A and B

Suite	**Config**	**Mean** $M_{i:j}$ $\in (0, 1]$	$M_{i:j} < 1$	**Coeff. Var.**
TACLe	No Cache Iso.	0.41	85%	0.59
	L3 Isolation	0.40	83%	0.59
	L2 + L3 Iso.	0.40	85%	0.53
DIS	No Cache Iso.	0.31	100%	0.57
	L3 Isolation	0.30	100%	0.66
	L2 + L3 Iso.	**0.34**	**100%**	**0.59**
SD-VBS	No Cache Iso.	0.52	95%	0.37
	L3 Isolation	0.56	97%	0.25
	L2 + L3 Iso.	0.52	95%	0.33

use a simpler M_i value that represents the maximum slowdown to τ_i when using SMT. Specifically, we define $M_i = \frac{\text{PET}_i^*}{\text{PET}_i}$, where PET_i^* is as defined immediately after Def. 2.

To calculate $M_{i:j}$, we measured PETs for all benchmark tasks and SMT combinations for three different configurations: *no cache isolation*, *L3 isolation*, and *both L2 and L3 isolation*. For L3 isolation, we allocated each task eight L3 cache ways, while for both L2 and L3 isolation, we allocated each task 16 L3 ways and half of the L2 colors, hence half of the L3 colors as well—refer again to Fig. 5. Note that, in both configurations, the total available L3 cache space is the same (we consider more restricted allocations later). We computed M_i values (which pertain to Level C) similarly, but did not assume any cache isolation (as Level-C tasks are provided no cache isolation from one another).

Table I summarizes the $M_{i:j}$ values we recorded for each benchmark suite and configuration by giving the frequency with which we saw $M_{i:j} < 1$—i.e., the percentage of task pairs that *benefit* from SMT—and the mean of $M_{i:j}$ values in the range (0, 1]—which provides a sense of the typical extent of improvement when there was improvement. We also include the mean of per-task-pair coefficients of variation (CV).[9] As an example, the row in bold indicates that, with both L2 and L3 isolation, all DIS task pairs benefit from SMT, with a typical per-pair $M_{i:j}$ value of 0.34 (recall that a smaller $M_{i:j}$ value indicates a higher benefit). Our full tables of both $M_{i:j}$ values and M_i values (which are not summarized here) are online.[7]

Obs. 1. *SMT is beneficial, even without any spatial isolation.*

This can be observed from Table I. Even without providing any cache isolation, we found the percentage of task pairs that benefit from SMT to be at least 85% (row 1, Table I), and the mean $M_{i:j}$ values of such tasks to be at most 0.52 (row 7).

Obs. 2. *Cache isolation minimally impacts SMT effectiveness.*

Columns 2 and 3 of Table I show this for each benchmark suite. This indicates that if a task uses a fixed-size exclusive partition of the L2 and L3 cache, no matter if it is paired or not, the cost of enabling SMT by adding a sibling task remains approximately the same. This implies that most of the slowdown attributed to SMT in prior work is due to execution unit interference rather than cache interference, and suggests that prior SMT effectiveness findings are valid even after cache isolation is added for predictability.

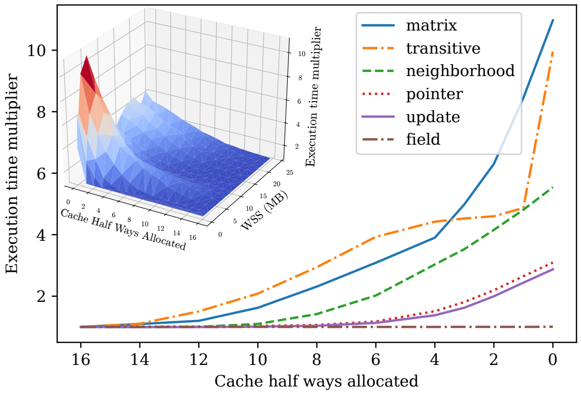

Fig. 7. DIS task execution times with a 4MB WSS as a function of allocated L3 cache space. The 3D inset shows corresponding data for the `matrix` benchmark for different WSSs.

C. Performance Effects of Cache Allocations

While cache isolation reduces interference, it decreases the maximum amount of cache available to a task. In the previous subsection, we considered the impacts of reduced interference, and here we consider the downsides of reduced cache space. Such downsides have been well-studied in prior work (e.g., see [9], [64]). Nonetheless, as our schedulability study in Sec. VI relies on execution-time and overhead measurements taken on our hardware platform, we conducted new experiments to assess the impacts of cache-space limits. We note also that such impacts have not been considered before in a system with dual L2/L3 cache management.

To investigate these impacts, we measured the execution times of all the DIS benchmarks in all WSS configurations for all potential L3 way allocations. We also used coloring to allocate half of the L2 cache. As a result, the L2 cache space was halved to 256KB and each allocated L3 way was halved to a 512KB *half-way*. We collected an extensive amount of data, most of which is available in our online appendix.[7] A sample of this data is given in Fig. 7, which depicts a 2D graph with a 3D inset. The 3D inset plots execution times for the `matrix` benchmark (y axis) as a function of the number of available L3 half-ways (x axis) and WSS (z axis). The 2D graph shows a slice of the 3D graph for all benchmarks at a WSS of 4MB.

Obs. 3. *With respect to allocated L3 cache space, tasks tend to fall into three categories:* no sensitivity, moderate sensitivity, *and* high sensitivity.

In Fig. 7, `field` exhibits no sensitivity (it tends to maintain L2 cache affinity), `update` and `pointer` exhibit moderate sensitivity, and the other benchmarks exhibit high sensitivity.

VI. Schedulability Study

To assess the benefits of adding SMT to MC^2 in a holistic sense, we conducted a large-scale overhead-aware

[9] The coefficient of variation is the standard deviation divided by the mean.

TABLE II
PARAMETERS USED IN SCHEDULABILITY STUDIES

Param.	Config.	Level A	Level B	Level C
Crit. Util. %	C-Heavy	[10,30]		[50,70]
	AB-Mod.	[35,45]		[10,30]
Task Util.	Light	[.001,.03]	[.001,.05]	[.001,.1]
	Medium	[.02,.1]	[.05,.2]	[.1,.4]
	Heavy	[.1,.2]	[.2,.4]	[.4,.6]
Period (ms)	Many	{5,10,20}	{20,40,80,160}	[10,100]
	Short	{3,6}	{6,12}	[3,33]
	Contrast	{3,6}	{96,192}	[10,100]
	Long	{48,96}	{96,192}	[50,500]
WSS (MB)	Default	$\langle 2.0, 2.0, .0032\rangle$		
Cache Sens.	Default	{1.16,2.95,15.68}		
SMT Effect.	DIS	$.00, \langle .34, .20, .01\rangle$		$\langle 1.60, .54, 1\rangle$
	TACLe	$.15, \langle .40, .21, .01\rangle$		$\langle 1.79, .32, 1\rangle$
	SD-VBS	$.05, \langle .52, .17, .01\rangle$		$\langle 1.72, .13, 1\rangle$
	TACLe+ SD-VBS	$.15, \langle .40, .21, .01\rangle$		$\langle 1.72, .13, 1\rangle$
	Prior	$.20, \langle .60, .07, .01\rangle$		$\langle 1.80, .10, 1\rangle$

$[a, b]$ denotes a uniform distribution; $\langle \mu, \sigma, \perp\rangle$ denotes a normal distribution centered at μ with standard deviation σ, where values less than $\perp$ are replaced with $\perp$ to prevent nonsensical (i.e., negative) values; $\{E\}$ denotes a random choice from the elements of set E.

schedulability study. In such a study, a large number of synthetic task systems are generated and their schedulability tested under different *schemes*. Schemes can represent different choices of analysis, heuristics, and hardware features (e.g., SMT) that impact schedulability. Such studies elucidate which schemes *generally* improve schedulability by aggregating the results of many task systems.

Synthetic task generation. Our schedulability-study framework is based on the open-source framework used by prior MC^2 projects found at [52]. Many of the initial steps taken in the task-generation process here are documented extensively in prior MC^2 works [8], [9], [18], [19], [24], [27]–[29].

Task generation is informed by the distributions in Table II. Some parameters listed here have different configuration options. Of particular relevance to our study are the Cache Sensitivity and SMT Effectiveness parameters. Cache Sensitivity determines how much the Level-A PET of each task inflates when given no L3 cache allocation. SMT Effectiveness gives $M_{i:j}$ values at Levels A and B, and M_i values at Level C. The distribution options for these parameters are based on the benchmark data shown in Fig. 7 and in Table I. A *scenario* is a selection of configuration for each parameter; considering different scenarios allows us to examine task-system-related tradeoffs. The detailed steps of the task-generation process are in Appendix A.

Assumptions. Models of synthetic tasks on multicore platforms inevitably require simplifying assumptions. For example, prior MC^2 schedulability studies have made assumptions about task interdependencies [8] and buffer-access patterns [29]. During task generation, we assumed that Cache Sensitivity and SMT Effectiveness are independent. In reality, a task's memory-access pattern affects both Cache Sensitivity and SMT Effectiveness (e.g., via memory stalls). From the data we have observed, ignoring such interdependencies in task generation does not cause large differences between synthetic PETs and corresponding benchmark measurements.

Evaluation methodology. We considered three schemes: Solo, which disallows threading (as in the preexisting MC^2 framework), Threaded, which threads tasks as per Sec. IV, and Solo+Threaded, which deems systems schedulable whenever either Solo or Threaded does so. These schemes have been sufficient to produce a rich set of observations from our schedulability study, though the consideration of alternative SMT-aware schemes would be useful in the future.

We considered both eight- and 16-core variants of our 3950X platform (two CCXs or four), and evaluated all scenarios for both variants. Our 16-core findings reinforce our eight-core findings, so we omit them here in favor of our online appendix.[7] In total, we considered 120 eight-core scenarios. For each scenario, scheme, and system utilization (from 0.5 to 1.5 times the core count), we measured the fraction of schedulable task systems within $\pm 5\%$ with 95% confidence. This took over 5,000 CPU hours (more than six CPU months).

For each scenario, we generated a schedulability graph indicating how the fraction of schedulable task systems varies per scheme (Solo+Threaded is not pictured as it is implicit from the other schemes). Graphs that highlight our observations are presented in Figs. 8-11 (additional graphs are online).[7] Note that system utilizations are graphed under Level-A pessimism and the usage of threading, so systems with utilization greater than the core count may be deemed schedulable.

To consider our schemes quantitatively, we use *Schedulable Utilization Areas (SUAs)*. For a single scenario, the SUA of a scheme is the area under its curve in the corresponding schedulability graph. This area is computed by the trapezoid rule. The SUA for a set of multiple scenarios is then the arithmetic mean of the SUAs of each scenario.

Results and observations. We now provide several observations that follow from the totality of our collected data. We illustrate these observations using the plots in Figs. 8-10.

Obs. 4. *Over all scenarios, the SUA of Solo+Threaded is 32% greater than that of Solo.*

A near-average scenario is shown in Fig. 8. The magnitude of this improvement is greater than that in prior work on SMT [3], which reported SUA improvements of around 19%. We believe this additional improvement is due to this prior work not considering overheads. While overheads clearly impact the schedulability of systems with or without SMT, the task-pairing heuristic we use to apply threading to Levels A and B effectively reduces the number of tasks to be scheduled, thereby reducing the corresponding scheduling overheads.

Obs. 5. *In most scenarios, the SUA of Solo+Threaded is 46% greater than that of Solo.*

This is backed by Fig. 9, where Threaded gives an SUA improvement of 46%. The SUA of Threaded is at least 46% greater than that of Solo in 55% of all scenarios. The DIS SMT Effectiveness distribution, as used in Fig. 9,

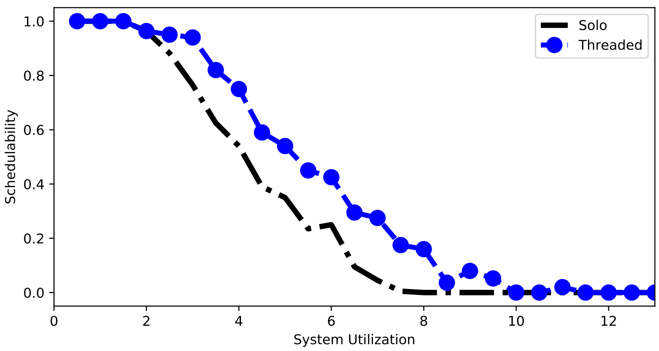

Fig. 8. AB-Moderate, Long, Moderate, DIS.

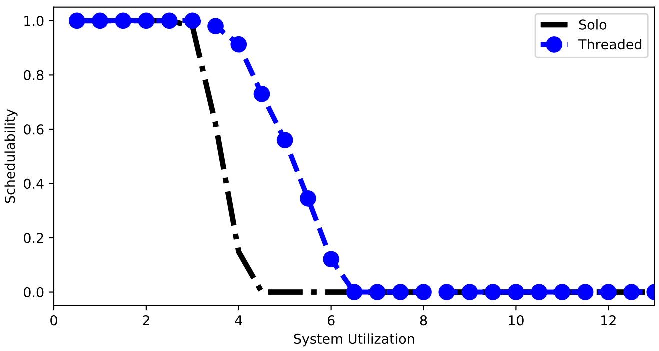

Fig. 9. AB-Moderate, Long, Light, DIS.

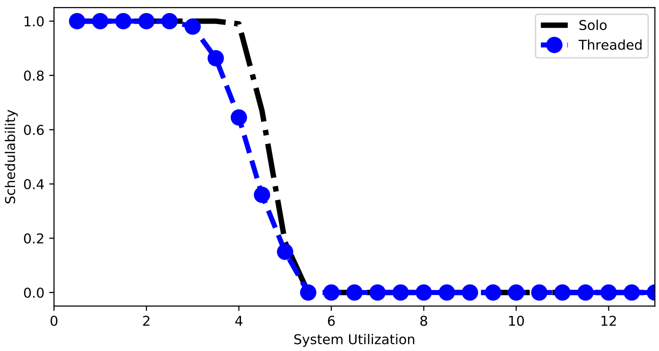

Fig. 10. C-Heavy, Long, Light, Prior.

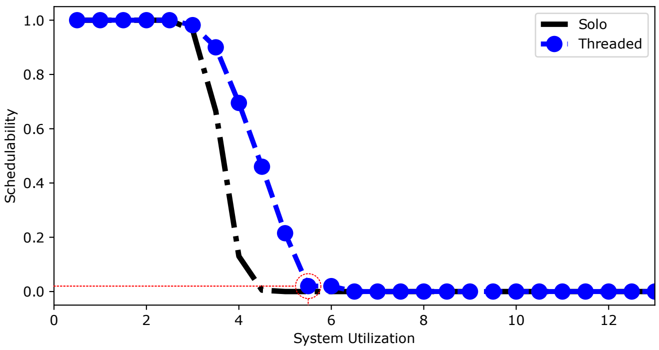

Fig. 11. AB-Moderate, Long, Light, TACLe+SD-VBS.

is beneficial to Threaded as the distribution has smaller SMT Effectiveness values than the others. Having light per-task utilizations and AB-moderate period distributions, like in Fig. 9, are additionally beneficial to Threaded. Since tasks with light utilizations are unlikely to exceed the framework's cap on threaded utilizations, light per-task utilizations afford more opportunities for using SMT, and the aforementioned reduction in overhead increases as more tasks are threaded.

Obs. 6. *In some scenarios, the SUA of Threaded is less than that of Solo.*

This is backed by Fig. 10, where Threaded gives a 10% reduction in SUA compared to Solo. Threaded has lower SUA than Solo in only 2.5% of all scenarios. Threaded is less effective for systems with high SMT Effectiveness, as with the Prior SMT Effectiveness distribution in 10. Threaded also tends to be less effective for the C-Heavy criticality distribution compared to the AB-moderate criticality distribution. This may because of the increased per-task utilizations which are penalized by our Level-C schedulability conditions.

Overall, SMT shows great promise for increasing the set of schedulable task systems on hardware that supports it; however, care must be taken such that SMT is not used in situations like those in Obs. 6. To avoid this, more advanced heuristics must be designed. Rather than just seeking to lower overall system utilization, such heuristics need to balance this objective against the fact that doing so may lead to increased capacity loss. Another option could be employing SMT, or not, on a per-level basis; perhaps the scenario in Fig. 10 would be better if SMT were used for Levels A and B, but not C.

VII. Case Studies

To validate the safe usage of SMT by our proposed approach and support the results of our schedulability study, we conducted case-study experiments. We created ten task systems for the scenario shown in Fig. 11 corresponding to the highest schedulable-utilization point in the Threaded curve (marked by lightly-dashed circle). Note that Solo has zero schedulability for this scenario. In each task system, we constructed Level-A and -B tasks from the TACLe benchmarks as representative of tasks having small runtimes and WSSs, and Level-C tasks from the SD-VBS benchmarks as representative of tasks with less-deterministic computations. On average, each task system has 562, 411, and 88 Level-A, -B, and -C tasks, respectively. We executed each task system for 60 minutes on both CCXs of our 3950X platform. Across all task systems, we observed no deadline misses by Level-A or -B tasks. Surprisingly, we did not observe deadline misses by the SRT Level-C tasks either. Across all task systems, 99.65% of the Level-A and -B tasks were paired, while 100% of the Level-C tasks were threaded. These results suggest the applicability of our approach to a real system.

VIII. Conclusion

We have presented an extension of the MC^2 framework that enables SMT-aware scheduling at all criticality levels. Introducing this awareness motivated providing dual L2/L3 shared-cache management. Through benchmarking experiments, we also examined how the use of SMT affects execution times in the presence of other interference sources. Based on data from these experiments and measured overheads on our 3950X platform, we conducted a large-scale overhead-aware schedulability study to assess the effectiveness of using SMT. In this study, SMT enabled average SUA gains of 32% and typical gains of over 46%. We reinforced these results by conducting case-study experiments involving actual running programs.

Due to space limitations, we focused our attention on the interference-mitigation mechanism shown to be of greatest impact in the pre-existing MC^2 framework, namely shared-cache management. However, the pre-existing MC^2 also provides other management features to mitigate interference due to DRAM conflicts, data sharing, I/O, etc. Future work will examine SMT alongside these features. Additionally, our 3950X platform enables a richer array of cache and bus-management options that warrant further consideration.

Appendix A
Task Generation for Schedulability Studies

We now list the steps for generating a task system τ. To aid in understanding these steps, we include Table II here with bolded entries as Table III and walk through an example of a Level-B task $\tau_i \in \tau$ being generated corresponding with those bold configurations. The steps of this example are given in bracketed comments. In the example, we denote the Level-L PET of task τ_i given W LLC ways as $\text{PET}_i^L(W)$, and the joint Level-L PET of paired task $\tau_{i:j}$ given W LLC ways as $\text{PET}_{i:j}^L(W)$.

1) Choose a goal total utilization U for τ, assuming Level-A pessimism for PETs. [*Let U of τ be 10.0.*]
2) Determine percent per-level contributions to U by sampling the relevant configuration for Criticality Utilization %. [*Suppose 40% of U of τ is from Level B. Then, the total utilization of Level B should be 4.0.*]
3) Sample the relevant configurations for Task Utilization, Period, and WSS[10] to generate tasks at each level. Briefly, these choices give, for each level, task utilizations (assuming Level-A pessimism and all 16 ways of L3 cache available), periods, and WSSs, respectively. Level-A PETs assuming a fully available L3 cache are computed by multiplying tasks' utilizations and periods. [*4.0 of task utilization must be generated for Level-B tasks. τ_i is such a task. Suppose its utilization, period, and WSS are chosen to be 0.3, 6 ms, and 4.0 MB. Then $PET_i^A(16) = 0.3 \cdot 6 = 1.8$ ms.*]

TABLE III
Parameters Used in Schedulability Studies

Param.	**Config.**	**Level A**	**Level B**	**Level C**
Crit.	C-Heavy	[10,30]		[50,70]
Util. %	**AB-Mod.**	[35,45]		[10,30]
Task	Light	[.001,.03]	[.001,.05]	[.001,.1]
Util.	Medium	[.02,.1]	[.05,.2]	[.1,.4]
	Heavy	[.1,.2]	[.2,.4]	[.4,.6]
Period	Many	{5,10,20}	{20,40,80,160}	[10,100]
(ms)	**Short**	{3,6}	{6,12}	[3,33]
	Contrast	{3,6}	{96,192}	[10,100]
	Long	{48,96}	{96,192}	[50,500]
WSS (MB)	Default	$\langle 2.0, 2.0, .0032\rangle$		
Cache Sens.	Default	{1.16,2.95,15.68}		
SMT	**DIS**	$.00, \langle .34, .20, .01\rangle$		$\langle 1.60, .54, 1\rangle$
Effect.	TACLe	$.15, \langle .40, .21, .01\rangle$		$\langle 1.79, .32, 1\rangle$
	SD-VBS	$.05, \langle .52, .17, .01\rangle$		$\langle 1.72, .13, 1\rangle$
	TACLe+ SD-VBS	$.15, \langle .40, .21, .01\rangle$		$\langle 1.72, .13, 1\rangle$
	Prior	$.20, \langle .60, .07, .01\rangle$		$\langle 1.80, .10, 1\rangle$

$[a, b]$ denotes a uniform distribution; $\langle \mu, \sigma, \perp\rangle$ denotes a normal distribution centered at μ with standard deviation σ, where values less than $\perp$ are replaced with $\perp$ to prevent nonsensical (i.e., negative) values; $\{E\}$ denotes a random choice from the elements of set E.

4) Sample Cache Sensitivity to determine how much the Level-A PET of each task inflates when given no L3 cache allocation. Level-A PETs for L3 allocations between full and none are interpolated by curves similar to prior MC^2 works (but using data points from our measurements) [19]. Such a curve depends on a task's WSS such that any allocated cache exceeding its WSS only marginally reduces its PET. [*Suppose τ_i's cache sensitivity is chosen to be 2.95. Then $PET_i^A(0) = 2.95 \cdot PET_i^A(16) = 2.95 \cdot 1.8 = 5.31$ms. τ_i's WSS is 4.0 MB (four L3 ways), so $\forall W \geq 4$: $PET_i^A(W) \approx PET_i^A(16)$. For all other W, $PET_i^A(W)$ is interpolated from these values.*]
5) Sample SMT Effectiveness for each task for each level to determine by how much PETs are inflated when running as a thread vs. as a solo task. For Levels A and B, a task's threaded PET is its solo PET added to M times the PET of its paired task. $M_i = \infty$ with probability equal to the fixed value in the cell in Table III of the current configuration option. Otherwise, M is drawn from the Normal distribution next to this value. This methodology follows [1], [3]. For Level-C tasks, threaded PETs are equal to solo PETs multiplied by the value sampled from the corresponding Normal distribution in Table III. [*Consider Level-B task τ_j. Then, with probability 0.68, $PET_{i:j}^L(W) = \infty$ for W ways and level L. Otherwise, suppose M_i is sampled to be 0.4. Then $PET_{i:j}^L(W) = PET_i^L(W) + 0.4 \cdot PET_j^L(W)$.*]

[10] In prior work on MC^2, an analogous parameter called Max Reload Time was used instead of WSS. Either parameter implicitly defines the other. We have chosen to make WSS explicit as we believe this increases clarity. We also inflated the distribution of WSSs for tasks compared to prior MC^2 works as our LLC is larger than that on the prior platform.

References

[1] S. Osborne and J. Anderson, "Simultaneous multithreading and hard real time: Can it be safe?" in *ECRTS*, 2020, pp. 14:1–14:25.

[2] S. Osborne, J. Bakita, and J. Anderson, "Simultaneous multithreading applied to real time," in *ECRTS*, 2019, pp. 1–22.

[3] S. Osborne, S. Ahmed, S. Nandi, and J. Anderson, "Exploiting simultaneous multithreading in priority-driven hard real-time systems," in *RTCSA*, 2020, pp. 1–10.

[4] A. Alhammad and R. Pellizzoni, "Trading cores for memory bandwidth in real-time systems," in *RTAS*, 2016, pp. 1–11.

[5] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *RTAS*, 2015, pp. 285–296.

[6] S. Altmeyer, R. Douma, W. Lunniss, and R. Davis, "Evaluation of cache partitioning for hard real-time systems," in *ECRTS*, 2014, pp. 15–26.

[7] N. Audsley, "Memory architecture for NoC-based real-time mixed criticality systems," in *WMC*, 2013, pp. 37–42.

[8] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, and F. D. Smith, "Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems," in *RTSS*, 2016, pp. 57–68.

[9] M. Chisholm, B. Ward, N. Kim, and J. Anderson, "Cache sharing and isolation tradeoffs in multicore mixed-criticality systems," in *RTSS*, 2015, pp. 305–316.

[10] G. Giannopoulou, N. Stoimenov, P. Huang, and L.Thiele, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *EMSOFT*, 2013, pp. 1–15.

[11] M. Hassan and H. Patel, "Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems," in *RTAS*, 2016, pp. 1–11.

[12] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems," in *RTAS*, 2015, pp. 307–316.

[13] J. Herter, P. Backes, F. Haupenthal, and J. Reineke, "CAMA: A predictable cache-aware memory allocator," in *ECRTS*, 2011, pp. 23–32.

[14] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and P. Cazorla, "A dual-criticality memory controller (DCmc) proposal and evaluation of a space case study," in *RTSS*, 2014, pp. 207–217.

[15] H. Kim, D. Broman, E. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based DRAM controller for mixed-criticality systems," in *RTAS*, 2015, pp. 317–326.

[16] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *RTAS*, 2014, pp. 145–154.

[17] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *ECRTS*, 2013, pp. 80–89.

[18] N. Kim, M. Chisholm, N. Otterness, J. Anderson, and F. D. Smith, "Allowing shared libraries while supporting hardware isolation in multicore real-time systems," in *RTAS*, 2017, pp. 223–234.

[19] N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," *Real-Time Systems*, vol. 53, no. 5, pp. 709–759, 2017.

[20] O. Kotaba, J. Nowotsch, M. Paulitsch, S. Petters, and H. Theiling, "Multicore in real-time systems – temporal isolation challenges due to shared resources," in *DATE*, 2013.

[21] Y. Krishnapillai, Z. Wu, and R. Pellizzoni, "ROC: A rank-switching, open-row DRAM controller for time-predictable systems," in *ECRTS*, 2014, pp. 27–38.

[22] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *DATE*, 2010, pp. 741–746.

[23] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric OS for multicore embedded systems," in *RTAS*, 2016, pp. 1–11.

[24] B. Ward, J. Herman, C. Kenna, and J. Anderson, "Making shared caches more predictable on multicore platforms," in *ECRTS*, 2013, pp. 157–167.

[25] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee, "Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation," in *RTAS*, 2016, pp. 1–12.

[26] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni, "PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms," in *RTAS*, 2014, pp. 155–166.

[27] J. Anderson, S. Baruah, and B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[28] M. Chisholm, N. Kim, S. Tang, N. Otterness, J. Anderson, F. D. Smith, and D. Porter, "Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems," in *RTNS*, 2017, pp. 58–67.

[29] N. Kim, S. Tang, N. Otterness, J. Anderson, F. D. Smith, and D. Porter, "Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks," in *RTNS*, 2018, pp. 191–201.

[30] N. Sritharan, A. Kaushik, M. Hassan, and H. Patel, "Enabling predictable, simultaneous and coherent data sharing in mixed criticality systems," in *RTSS*, 2019, pp. 433–445.

[31] S. Cheng, J. Chen, J. Reineke, and T. Kuo, "Memory bank partitioning for fixed-priority tasks in a multi-core system," in *RTSS*, 2017, pp. 209–219.

[32] J. Xiao, S. Altmeyer, and A. Pimentel, "Schedulability analysis of non-preemptive real-time scheduling for multicore processors with shared caches," in *RTSS*, 2017, pp. 199–208.

[33] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing mixed criticality applications on modern heterogeneous mpsoc platforms," in *ECRTS*, 2019, pp. 27:1–27:25.

[34] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *ECRTS*, 2012, pp. 299–308.

[35] ——, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS*, 2013, pp. 55–64.

[36] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *RTAS*, 2016, pp. 1–12.

[37] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *RTAS*, 2013, pp. 45–54.

[38] 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 *RTAS*, 2019, pp. 1–14.

[39] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *RTAS*, 2011, pp. 269–279.

[40] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *RTAS*, 2020, pp. 239–252.

[41] M. Xu, L. Thi, X. Phan, H. Choi, and I. Lee, "vCAT: Dynamic cache management using cat virtualization," in *RTAS*, 2017, pp. 211–222.

[42] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *RTAS*, 2019, pp. 345–356.

[43] M. Hassan and R. Pellizzoni, "Analysis of memory-contention in heterogeneous COTS mpsocs," in *ECRTS*, 2020, pp. 23:1–23:24.

[44] J. Rivas, J. Goossens, X. Poczekajlo, and A. Paolillo, "Implementation of memory centric scheduling for COTS multi-core real-time systems," in *ECRTS*, 2019, pp. 7:1–7:23.

[45] M. Hassan, "Discriminative coherence: Balancing performance and latency bounds in data-sharing multi-core real-time systems," in *ECRTS*, 2020, pp. 16:1–16:24.

[46] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "Wcet(m) estimation in multi-core systems using single core equivalence," in *ECRTS*, 2015, pp. 174–183.

[47] B. Ocker, "FAA special topics," in *Collaborative Workshop: Solutions for Certification of Multicore Processors*, Nov. 2018.

[48] *Neoverse E1 Core Technical Reference Manual*, ARM, 2019, rev. r1p1.

[49] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, pp. 12–19, 1997.

[50] J. Bulpin, "Operating system support for simultaneous multithreaded processors," Ph.D. dissertation, University of Cambridge, King's College, 2005.

[51] J. Bulpin and I. Pratt, "Multiprogramming performance of the Pentium 4 with hyperthreading," in *The Third Annual Workshop on Duplicating, Deconstruction and Debunking*, 2004, pp. 53–62.

[52] "LITMUS^{RT} home page," Online at http://www.litmus-rt.org/, 2020.

[53] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. Anderson, "LITMUS^{RT} : A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS*, 2006, pp. 111–126.

[54] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2011.

[55] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *RTSS*, 2007, pp. 239–243.

[56] J. Liedtke, H. Härtig, and M. Hohmuth, "OS-controlled cache predictability for real-time systems," in *RTAS*. IEEE, 1997, pp. 213–224.

[57] *AMD64 Technology Platform Quality of Service Extensions*, ARM, 2018, rev. 1.00.

[58] G. Jia, X. Li, Y. Yuan, J. Wan, C. Jiang, and D. Dai, "Pseudonuma for reducing memory interference in multi-core systems," in *HPC*, 2014, pp. 1–8.

[59] M. Hillenbrand, M. Gottschlag, J. Kehne, and F. Bellosa, “Multiple physical mappings: Dynamic DRAM channel sharing and partitioning,” in *the 8th Asia-Pacific Workshop on Systems*, 2017, pp. 1–9.

[60] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos, “Mixed criticality real-time scheduling for multicore systems,” in *ICESS*, 2010, pp. 1864–1871.

[61] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, “Taclebench: A benchmark collection to support worst-case execution time research,” in *WCET*, 2016, pp. 1–10.

[62] A. A. Division, “DIS Stressmark Suite,” Titan Systems Corporation, Tech. Rep., 2000, ver. 1.0.

[63] S. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. Taylor, “SD-VBS: the san diego vision benchmark suite,” in *IISWC*, 2009, pp. 55–64.

[64] B. Bui, M. Caccamo, L. Sha, and J. Martinez, “Impact of cache partitioning on multi-tasking real time embedded systems,” in *RTCSA*, 2008, pp. 101–110.