Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning *

Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

The multicore revolution is having limited impact in safety-critical application domains. A key reason is the "one-out-of-m" problem: when validating real-time constraints on an m-core platform, excessive analysis pessimism can effectively negate the processing capacity of the additional $m-1$ cores so that only "one core's worth" of capacity is available. Two approaches have been investigated previously to address this problem: mixed-criticality allocation techniques, which provision less-critical software components less pessimistically, and hardware-management techniques, which make the underlying platform itself more predictable. A better way forward may be to combine both approaches, but to show this, fundamentally new criticality-cognizant hardware-management tradeoffs must be explored. Such tradeoffs are investigated herein in the context of a large-scale, overhead-aware schedulability study. This study was guided by extensive trace data obtained by executing benchmark tasks on a new variant of the MC^2 framework that supports configurable criticality-based hardware management. This study shows that the two approaches mentioned above can be much more effective when applied together instead of alone.

1 Introduction

Multicore platforms have the potential of enabling a wealth of new computationally intensive features in safety-critical domains such as in the avionics and automotive industries. However, certifying the real-time correctness of a system running on m cores can require analysis that is so pessimistic, the processing capacity of the additional $m-1$ cores is entirely negated. In effect, only "one core's worth" of capacity can be utilized even though m cores are available. In domains such as avionics, this "one-out-of-m" problem has led to the common practice of simply disabling all but one core.[1] This problem is the most serious unresolved obstacle in work on real-time multicore resource allocation today.

The root of this problem is that shared hardware resources, such as caches, buses, and memory banks, are not predictably managed. As reviewed later, several proposals for predictably managing such resources have been presented that strive to reduce pessimism by enabling tighter task execution-time estimates. While these approaches seem promising, another way forward is the application of *mixed-criticality* (*MC*) analysis assumptions, as originally proposed by Vestal [28]. Under such assumptions, less-critical tasks are provisioned somewhat optimistically, allowing for increased platform utilization. These research directions share a similar goal of improving platform utilization, but are themselves orthogonal, raising broader research questions pertaining to the combination of both approaches. Can better platform utilization be realized if resources are managed differently at different criticality levels? If so, how should resources be managed both within and across criticality levels?

Isolation versus sharing. Addressing these questions requires delving into sharing/isolation tradeoffs that have not been considered before. For example, while higher-criticality tasks might require strong hardware-isolation guarantees, more optimistically provisioned lower-criticality tasks might actually *benefit* from less restricted hardware sharing because shared hardware is often designed to improve average-case performance or throughput. With respect to caches, higher-criticality tasks might tolerate severe restrictions on cache usage, because they are provisioned pessimistically anyway. For lower-criticality tasks, the opposite may be true.

In this paper, we report on our efforts to construct an experimental platform that enables such tradeoffs to be assessed, and discuss the results of an experimental investigation conducted to provide such an assessment. Our new platform extends a framework called MC^2 (mixed-criticality on multicore) [11, 24, 29], which has been the subject of continuing research by our group, by adding support for several hardware-management techniques. Specifically, we provide management for both the *last-level cache* (*LLC*) and *DRAM memory banks*. Additionally, we provide techniques that isolate the operating system (OS) from user-space tasks with respect to the LLC and DRAM banks; to our knowledge, the issue of OS isolation has not been considered before in work on hardware management. We regard MC^2 as a rich and interesting platform for our investigation because (as discussed later) it supports several criticality levels (not just two, as typically assumed in work on MC scheduling), has both hard real-time (HRT) and soft real-time (SRT) components, both priority-scheduled and time-triggered components, and both partitioned and globally scheduled components.

Contributions. Our contributions are threefold. First, after providing needed background (Sec. 2), we describe the hardware-management mechanisms we added to MC^2 (Sec. 3). The resulting MC^2 variant is highly configurable and breaks new ground by allowing sharing/isolation trade-

*Work supported by NSF grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and a grant from General Motors. The second author was also supported by an NSF graduate fellowship.

[1] Multicore-related certification difficulties are extensively discussed in a recent position paper from the U.S. Federal Aviation Administration [7].

offs to be studied in a criticality-cognizant way.

Second, we present extensive experiments concerning such tradeoffs that demonstrate the value of managing hardware in MC systems (Sec. 4.1). In the considered MC^2 configuration, strong hardware-isolation guarantees were provided to highly critical tasks, but somewhat permissive sharing was allowed for less-critical tasks.

Third, we provide evidence in favor of combining MC analysis with hardware management in attacking the one-out-of-m problem. This evidence is provided in the form of a large-scale overhead-aware schedulability study (Sec. 4.2) that we conducted to demonstrate the benefits of combining both approaches, and associated runtime experiments that we conducted to partially assess the reasonableness of some of the assumptions underlying this study (Sec. 4.3). To our knowledge, this is the first overhead-aware schedulability study that considers MC scheduling, hardware management, and a combination of both.

In work on MC systems, there has been some limited prior work in which hardware management was applied (see Sec. 2). However, to our knowledge, we are the first to provide criticality-aware isolation—with respect to both the OS and some of the most problematic sources of hardware interference—within a framework as diverse as MC^2, under *Vestal's notion of MC analysis*, which was proposed with the express goal of *improving platform utilization*.

2 Background

We begin by reviewing needed background and related work.

Task model. We consider real-time workloads specified using the implicit-deadline *periodic task model* and assume familiarity with this model. We specifically consider a task system $\tau = \{\tau_1, \ldots, \tau_n\}$, scheduled on m processors,[2] where task τ_i's *period* and *worst-case execution time* (*WCET*) are denoted T_i and C_i, respectively. (We generalize this model below when considering MC scheduling.) The *utilization* of task τ_i is given by $u_i = C_i/T_i$ and the *total system utilization* is defined as $\sum_i u_i$. A periodic task system may be scheduled following a *partitioned approach* (tasks are statically assigned to processors), a *global scheduling* approach (any task may execute on any processor), or some hybrid of the two. If a job of τ_i has a deadline at time d and completes execution at time t, then its *tardiness* is $max\{0, t - d\}$. Tardiness should be zero for any job of a *hard real-time* (*HRT*) task, and should be bounded by a (reasonably small) constant for any job of a *soft real-time* (*SRT*) task.

Mixed-criticality scheduling. The roots of most recent work on MC scheduling can be traced to a seminal paper by Vestal [28]. For systems where tasks of differing criticalities exist, he proposed adopting less-pessimistic execution-time assumptions when checking the schedulability of less-critical tasks. More formally, in a system with L criticality levels, each task has a *provisioned execution time* (*PET*)[3] specified at every level, and L system variants are analyzed: in the Level-ℓ variant, the real-time requirements of all Level-ℓ tasks are verified with Level-ℓ PETs assumed for *all* tasks (at any level). The degree of pessimism in determining PETs is level-dependent: if Level ℓ is of higher criticality than Level ℓ', then Level-ℓ PETs will generally be greater than Level-ℓ' PETs. For example, in the systems considered by Vestal [28], observed WCETs were used to determine PETs for tasks at lower levels, and such times were inflated to determine PETs at higher levels. The task model resulting from Vestal's work has come to be known as the *MC task model*.

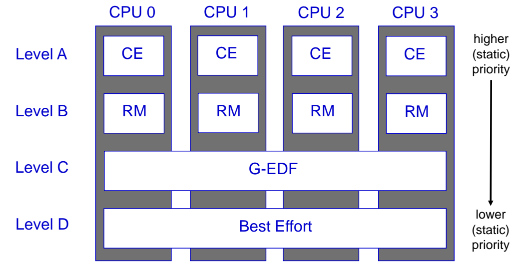


Figure 1: Scheduling in MC^2 on a quad-core machine.

MC^2. Vestal's work led to a significant body of follow-up work (see [5] for an excellent survey). Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors (to our knowledge) [11, 24, 29]. MC^2 is implemented as a LITMUS^{RT} [22] plugin and supports four criticality levels, denoted A (highest) through D (lowest), as shown in Fig. 1. Higher-criticality tasks are statically prioritized over lower-criticality ones. Level-A tasks are partitioned and scheduled on each core using a time-triggered table-driven cyclic executive.[4] Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each core.[4] On each core, the Level-A and -B tasks are required to be simply periodic (all tasks commence execution at time 0 and periods are harmonic), with the Level-B task periods being integer multiples of the Level-A hyperperiod. Level-C tasks are scheduled via a global earliest-deadline-first (GEDF) scheduler.[4] Level-A and -B tasks are HRT, Level-C tasks are SRT, and Level-D tasks are non-real-time (so we do not consider them further). MC^2 is a flexible framework. For example, it can be configured to have only two HRT criticality levels (as in most theoretical work on MC scheduling) or to fully assign the Level-A and -B subsystems to distinct, dedicated cores.

MC^2 was originally designed in consultation with colleagues in the avionics industry. A major thesis underlying its design is that Levels A and B would be mostly comprised of quite deterministic "fly-weight" tasks with rather low utilizations; less-deterministic computationally intensive tasks of higher utilization would likely be assigned to Level C.

Page coloring. Page coloring is a technique that can be applied to eliminate interference within the LLC and memory

[2]We use the terms "processor," "core," and "CPU" interchangeably.

[3]We use "PET" instead of "WCET" because under MC^2, some tasks are SRT, and hence may not be provisioned on a worst-case basis.

[4]Other per-level schedulers optionally can be used, and Level-C tasks can be defined according to the sporadic task model. These options, and other considerations, such as slack reallocation, schedulability conditions, and execution-time budgeting are discussed in prior papers [11, 24, 29].

banks [14]. We explain the basic idea here with respect to the LLC (which we assume to be set-associative). Consider the pages of physical memory in turn. Assign the color "0" to the first page, and assign the same color to the sets in the LLC to which its content's addresses map. In a similar way, assign the color "1" to the next page and corresponding cache sets, and so on. Eventually, such color assignments will "wrap" and we will sequence through the same colors again. This process ensures that differently colored pages map to different sets in the LLC. Thus, accesses to two differently colored pages cannot cause cache conflicts. Note that this coloring process is based on physical memory addresses. Such addresses also determine how memory pages map to DRAM banks, so pages can also be colored with respect to the banks to which they are mapped.

Ensuring isolation with respect to the LLC. In most prior work on eliminating or reducing interference in the LLC, some variant of cache partitioning is used (see [19] for an overview). *Set-based* cache partitioning can be implemented by page coloring: each partition corresponds to a disjoint subsequence of colors that maps to some disjoint subsequence of sets in the LLC. *Way-based* cache partitioning is also possible, but this requires hardware support. The ARM platform utilized in our experiments provides such support, which we describe in detail later in Sec. 3 (see Fig. 3).

Ensuring isolation with respect to memory banks. Modern DRAM designs contain multiple banks, which can be interleaved to parallelize memory accesses. Each bank consists of memory in an array of rows and columns, along with a row buffer. For a memory location to be read or written via the data bus, that location's row must be stored in the row buffer. If the row was already in the buffer, then we have a *row-buffer hit*, otherwise we have a *row-buffer miss*. In the event of a miss, the row previously in the buffer must be copied back to the array. Row-buffer misses create extra latency. Tasks executing on different processors can be prevented from causing each other to experience such misses by partitioning DRAM banks among processors [23].

Prior related work. The use of cache partitioning in real-time systems has been considered by Kim *et al.* [17] and Altmeyer *et al.* [2]. However, both of these papers are directed at uniprocessor platforms and neither considers MC systems. As an alternative to cache partitioning, a technique called *cache lockdown* can be used that prevents designated cached data or instructions from being evicted [6]. Also, it is possible to redesign the cache allocator itself to provide a replacement policy that is more predictable [12].

Regarding memory-related issues generally, prior work has been done on more predictable memory architectures and memory controllers for single-criticality [21] and MC [3, 15, 26] systems, on improved timing analysis for MC multicore systems that more accurately assesses memory interference [13], and on making bus accesses more predictable in single-criticality [1, 27] and MC systems [9, 10]. Regarding DRAM-related issues specifically, Kim *et al.* [16] presented analysis for predicting memory-access delays in which DRAM characteristics are carefully considered, and

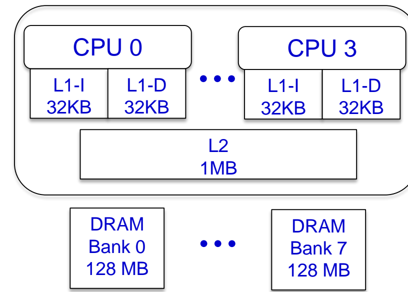


Figure 2: Quad-core ARM Cortex A9.

Yun *et al.* [30] presented PALLOC, an OS-based dynamic memory allocator that can specifically allocate pages to provide bank and/or cache isolation. Neither of these papers considers MC systems. In other cache-related work, Yun *et al.* [31] presented an approach that reduces cache, bus, and memory interference in MC systems by stalling some memory accesses. A survey of challenges created by shared hardware interference has been presented by Kotaba *et al.* [20].

Of the just-cited papers, only three consider the notion of MC scheduling espoused by Vestal [3, 13, 15]. These three papers focus on hardware issues and only peripherally touch on the issue of *achieving better platform utilization from a schedulability point of view through less pessimistic analysis assumptions* as proposed by Vestal. This issue is the subject of two prior MC^2-related papers by our group. The first of these papers [29] considers a scheduling-based approach to LLC management for high-criticality tasks only. In that work, the OS prefetches all potentially accessed pages before a high-criticality job executes, and enforces that co-scheduled jobs do not conflict in the LLC. Lower-criticality jobs are allowed to execute in periods of high LLC contention that otherwise would have idled a processor. In contrast, we take a more holistic approach to hardware management here, and consider hardware-management tradeoffs within and among all criticality levels. The second paper [8] tackles the theoretical problem of assigning different portions of the LLC to different criticality levels by applying a combination of set- and way-based partitioning. This last paper is actually a stepping stone towards the work described herein.

Specifically, we build upon that paper by considering OS and DRAM-bank isolation, by analyzing important overhead sources based on an actual implementation, and by applying this overhead data in a large-scale schedulability study. Our overall research agenda breaks new ground on several fronts. First, we are the first to investigate sharing/isolation tradeoffs with respect to shared hardware in a criticality-cognizant way (prior work only emphasized isolation). Second, we consider systems with more than two criticality levels (as opposed to only two, as in almost all prior work). Third, we are the first to investigate cache partitioning in MC systems, and in systems in which both partitioned and global schedulers are used. Fourth, we are the first to consider interference with respect to both the LLC and DRAM memory banks in MC multicore systems. Finally, we are the first to address shared-hardware interference due to the OS.

3 Implementation

We now describe the hardware-management extensions we added to MC[2]. All source code for our new MC[2] prototype is available online [22]. To discuss the specific hardware resources to be managed, we must first describe our considered hardware platform, which is a quad-core ARM Cortex A9 machine. Each core on this machine is clocked at 800MHz and has separate 32KB L1 instruction and data caches. Additionally, the LLC is a shared, unified 1MB 16-way set-associative L2 cache. 1GB of off-chip DRAM is available, and this memory is partitioned into eight 128MB banks. The basic architecture is illustrated in Fig. 2.

Way- and set-based LLC partitioning. Our ARM platform provides per-CPU *lockdown registers* that enable the LLC to be partitioned by way. This is illustrated in Fig. 3(a). In the depicted situation, the lockdown bit corresponding to Way 2 is cleared on CPU 0, which directs cache allocations from CPU 0 to Way 2 of the LLC. Per-CPU lockdown registers can be modified via the *proc* filesystem interface.

As an alternative to way-based partitioning, our implementation allows set-based partitioning via page coloring. This is illustrated in Fig. 3(b) for our ARM platform, which has an LLC with 16 colors. Way- and set-based partitioning can be combined to flexibly create rectangular LLC areas that can be designated for the sole use of certain tasks. This is illustrated in Fig. 3, which depicts our overall allocation strategy; we consider this figure in detail later.

DRAM banks. Our test platform allows DRAM bank interleaving to be optionally enabled. With bank interleaving enabled, successive pages map to different banks; with it disabled, the first 32K pages map to Bank 0, the next 32K to Bank 1, and so on. Bank interleaving results in increased memory throughput in certain use cases. However, when enabled on our test platform, the bits within a physical address that determine the mapped-to bank overlap those that determine the LLC color, and as a result, each bank contains pages of only two LLC colors. In contrast, with interleaving disabled, each bank contains pages of all LLC colors. The latter permits more fine-grained control over page allocations, so we disable bank interleaving. However, when allocating pages to tasks, we attempt to distribute a task's pages across all of the banks that it can access (if more than one), to obtain the benefits of bank interleaving. The manner in which we allocate pages is discussed next.

Allocating pages to tasks. A memory location's physical address determines both its LLC color and DRAM bank. To properly allocate LLC colors and DRAM banks to tasks, we construct pools of pages for each color and bank combination. We then reallocate pages to tasks from these pools via a system call. In our experiments, we were able to fully allocate to these pools all pages from four of the DRAM banks, Banks 3 through 6. Each of these four banks is dedicated to the Level-A and -B tasks on a specific CPU of our quad-core platform. The other four banks are shared by the OS and all Level-C tasks. As a result, the OS can allocate pages only from these banks and dynamic memory allocation is

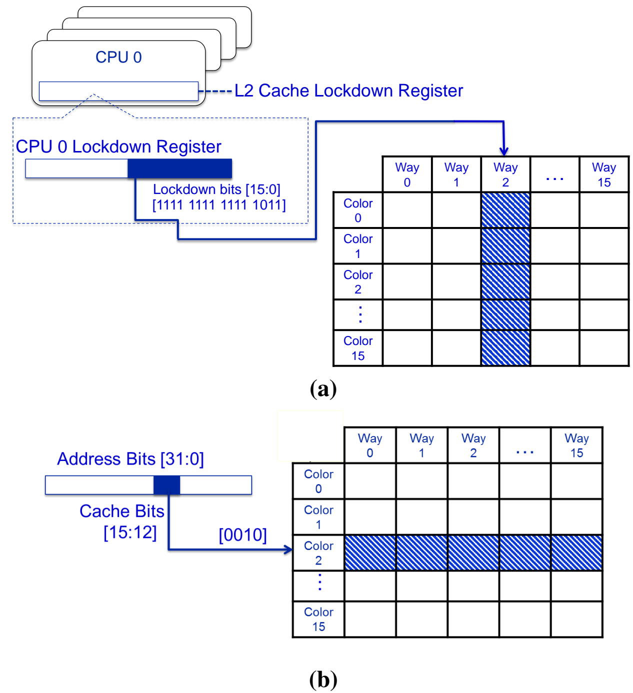


Figure 3: LLC partitioning **(a)** by way and **(b)** by set.

only supported for Level-C tasks.[5] With the exception of a rarely accessed signal-handling page, our page-coloring process can color all pages associated with each task. However, we currently do not allocate shared pages, though shared libraries can be dealt with via static linking. We defer full consideration of shared pages to future work.

Way-based OS isolation. Our prototype isolates the OS from Level-A and -B tasks in the LLC via way-based partitioning. Specifically, whenever kernel code begins executing on a CPU as the result of an interrupt, exception, or system call, we save the current value of that CPU's lockdown register and then modify it so that the OS code accesses only certain LLC ways in kernel mode. When exiting kernel mode, we restore the lockdown register using the saved value. Together with the DRAM isolation just described, this ensures that the OS only minimally interferes with Levels A and B.

Unmanaged hardware resources. Our prototype does not provide management for L1 caches, translation lookaside buffers (TLBs), memory controllers, or memory buses. However, we assume a measurement-based approach to determining PETs, so such unconsidered resources are implicitly considered when PETs are determined. We adopt a measurement-based approach because work on static timing-analysis tools for multicore machines has not matured to the point of being directly applicable. Moreover, measurement-based processes for determining PETs are often used in practice.

Overall allocation strategy. Our overall allocation strategy is depicted in Fig. 4. This strategy ensures strong isolation guarantees for higher-criticality tasks, while allowing for

[5] According to the thesis underlying the design of MC[2] (mentioned in Sec 2), Level-A and -B tasks are expected to be fly-weight, deterministic tasks, and hence should not require dynamic memory allocation.

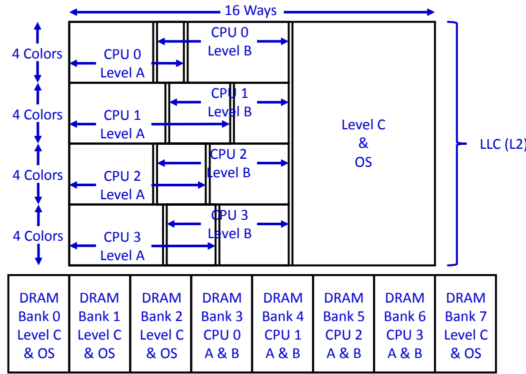


Figure 4: Example LLC and DRAM allocation. Note that the Level-A and -B LLC areas for each core can overlap. LLC boundaries indicated by double lines are settable parameters.

fairly permissive hardware sharing for lower-criticality tasks. DRAM allocations are depicted at the bottom of the figure and are as discussed earlier. LLC allocations, which we describe next, are depicted at the top.

As seen, Level C and the OS share a subsequence of the available LLC ways and all LLC colors. In prior work on MC^2 [11, 24, 29], Level-C tasks (being SRT) were assumed to be provisioned on an average-case basis, and we assume that here. Under this assumption, LLC sharing with the OS should not be a major concern. The remaining LLC ways are partitioned among Level-A and -B tasks on a per-CPU basis. That is, the Level-A and -B tasks on a given core share a partition. Each of these partitions is allocated 1/4 of the available colors, as depicted. This scheme ensures that Level-A and -B tasks do not experience LLC interference due to tasks on other cores (*spatial* isolation). Also, Level-A tasks (being of higher priority) do not experience LLC interference due to Level-B tasks on the same core (*temporal* isolation).

The specific number of LLC ways allocated to the Level-C/OS partition and to the per-core Level-A and -B partitions is a tunable parameter determined on a per-task-set basis using optimization techniques that were the main contribution of the precursor paper to this one [8]. These optimization techniques seek to minimize a task set's Level-C utilization while ensuring schedulability at all criticality levels.

In addition to the basic allocation strategy depicted in Fig. 4, we also consider a variant of it in which the Level-C/OS LLC area is partitioned by way on a per-CPU basis. This variant provides stronger isolation guarantees to Level-C tasks but reduces the LLC area that such a task can utilize.

4 Evaluation

We experimentally assessed the impact of combining MC allocation and hardware management in attacking the one-out-of-m problem via the following process. First, to assess the impact of hardware management, we collected extensive trace data concerning synthetic tasks and benchmark programs. A subset of this data is discussed in Sec. 4.1. Second, we conducted a large-scale overhead-aware schedulability study involving task systems randomly generated using a process based on our collected trace data. A subset of these experimental results is presented in Sec. 4.2. In these schedulability experiments, certain provisioning assumptions were made regarding PETs. To partially assess the soundness of these assumptions, we conducted experiments in which runtime data was collected for systems of benchmark programs. This data is presented and discussed in Sec. 4.3.

Programs	Memory Access
Pointer	Small blocks at unpredictable locations.
Update	Small blocks at unpredictable locations with memory updates.
Matrix	Irregular or mixed, with mixed levels of reuse.
Neighborhood	Regular access to pairs of words at arbitrary distance.
Field	Regular, with little reuse.
Transitive Closure	Reads and writes to different matrices concurrently.

Table 1: DIS Stressmark programs [25].

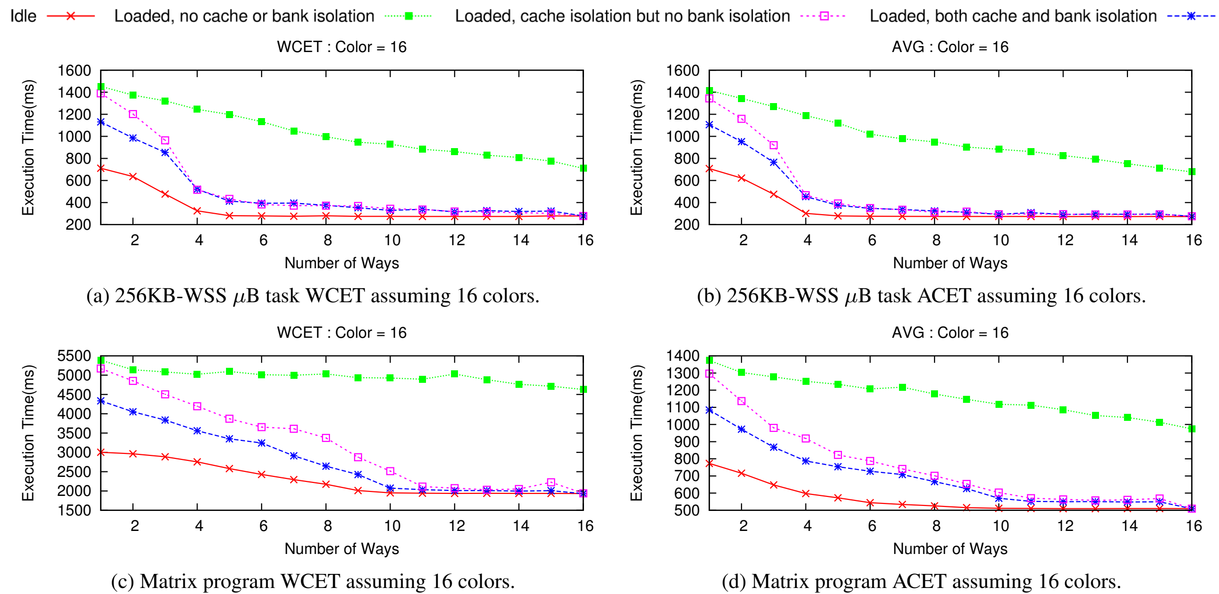
4.1 Isolation Impacts

We examined isolation impacts by collecting trace data for both synthetic micro-benchmark (μB) tasks devised by us and publicly available benchmark programs. The μB tasks were designed as stress cases to demonstrate the upper limits of potential performance improvements made possible by LLC and DRAM bank management. Each μB task consists of a main loop that is repeated 500 times. During each loop iteration, a different randomly chosen sequence of unique word addresses is read, where each address aligns with the first word in a cache line (32 bytes on our hardware). Every available cache line is referenced once in an iteration. This access pattern has the effect of forcing each cache reference to a random line and eliminating hits for successive references within a line (reducing spatial and temporal locality in references). Each μB task has a specified *working set* (*WS*), which is the set of addresses used to reference data, and correspondingly a *working set size* (*WSS*).

The benchmark programs we considered are listed in Tbl. 1 and come from the *Data Intensive Systems (DIS) Stressmark Suite* [25]. This suite was defined to reflect memory-usage patterns common in real-world use cases.

Quantifying cache-usage patterns is harder for real application code than μB tasks. However, there must naturally be a point of diminishing returns for larger and larger LLC allocations for any task (assuming it is executed in isolation). We call this point, where execution times do not substantially decrease given a larger LLC allocation, a task's *ideal cache allocation size* (*ICAS*). Note that it is possible for a task to have an ICAS larger than the LLC. In such cases, we define its ICAS to match the LLC size. Our μB tasks have a very small code footprint, so for them, ICAS is the same as WSS.

Impact of providing full isolation. In our first set of experiments, we examined the impact of providing full isolation to a task by giving it a dedicated LLC area and/or DRAM bank that are accessed by no other task. When full isolation is provided, these experiments have implications when determining PETs for Level-A and -B tasks. Such tasks can only experience interferences from other tasks due to preemptions, and any execution-time increases due to preemptions



(a) 256KB-WSS μB task WCET assuming 16 colors.

(b) 256KB-WSS μB task ACET assuming 16 colors.

(c) Matrix program WCET assuming 16 colors.

(d) Matrix program ACET assuming 16 colors.

Figure 5: Execution-time data for **(a) & (b)** the 256KB-WSS μB task and **(c) & (d)** the Matrix program.

are dealt with in overhead accounting.

To assess the impacts of providing full isolation, we ran experiments in which a measured task (either a μB task or a DIS program) was run alone on one core either in the presence of no other running tasks—we call this the *idle scenario*—or along with *stress-inducing tasks* running concurrently on the other three cores—we call this the *loaded scenario*. The loaded scenario was further factored into three cases: **(i)** *no cache or bank isolation*, **(ii)** *cache isolation but no bank isolation*, and **(iii)** *both cache and bank isolation*. This yielded a total of four isolation configurations. The stress-inducing tasks were configured like our synthetic μB tasks, with a WSS of 1MB. When bank isolation was not provided, these tasks were configured to specifically target the DRAM bank used by the measured task.

For each measured task and isolation configuration, we considered 272 possible LLC area sizes (given by 0 to 16 ways and 1 to 16 colors) for allocation to the measured task. Each additional way or color increases the allocated LLC space by 4KB. This process yielded $272 \times 4 = 1{,}088$ experiments per measured task. In each such experiment, we recorded the (observed) WCET and average-case execution time (ACET) of the measured task from data collected in 100s of execution. We are interested in both WCETs and ACETs because we assume that Level-A and -B PETs are based on WCETs, and Level-C PETs are based on ACETs, as in prior work on MC^2 [11, 24, 29].

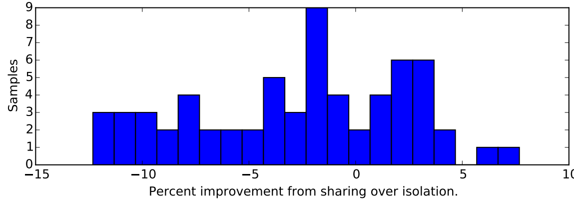
We collected trace data (6GB in total) for all six programs in Tbl. 1 and for μB-task WSSs in $\{32, 64, 256, 512\}$KB. We now make several observations based on this data. We support these observations using the data in Fig. 5, which depicts recorded WCET and ACET values for both the 256KB-WSS variant of the μB task and the Matrix program, given an allocated LLC area consisting of 16 colors and some number of ways. Due to space constraints, the rest of our collected data is given in an online appendix [18].

Obs. 1. For a given LLC allocation, isolation reduced WCETs by up to 277% for the μB task and by up to 242% for the Matrix program.

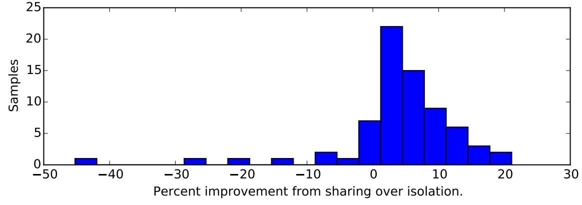
The noted 277% (resp., 242%) reduction can be seen by comparing the curves in Fig. 5(a) (resp., Fig. 5(c)) at the data points corresponding to five (resp., twelve) ways. However, isolation comes at a cost. For example, if we choose to isolate Level-C tasks by way on a per-CPU basis (refer to Fig. 4), then each Level-C task would only be able to access 1/4 of the available LLC area size (assuming it is divisible by four) instead of sharing the entire area.

Obs. 2. For a given LLC allocation space, sharing the entire space without isolation and partitioning the space and providing isolation are incomparable with respect to ACETs and WCETs. This exposes a tradeoff that is dependent upon the given application, as well as the size and configuration of the LLC-allocation space under consideration.

For example, suppose we have the option at Level C of either sharing the entire LLC among all four cores or providing per-core partitions with four ways and 16 colors. From the perspective of the μB task considered in Fig. 5(b), these two options can be compared by examining the curves for a loaded system with LLC isolation at four ways and no LLC isolation at 16 ways. From these two points, we see that the ACETs for these two options are $466\,\mathrm{ms}$ and $677\,\mathrm{ms}$, respectively, a 31% ACET improvement under isolation. To more clearly investigate this tradeoff, we generated histograms, two of which are given in Fig. 6, that visually depict the distribution of how frequently, and by how much, sharing vs. isolation improves ACETs and WCETs. The two presented histograms show that for the Matrix program, isolation tends to improve ACETs, while for the 256KB-WSS μB task, the reverse is true. Also, there exist cases in which one approach may be substantially better, as demonstrated by the leftmost "outlier" in Fig. 6(b), where isolation yields a 45% improvement. Given this tradeoff, we opted to fully consider in our



(a) Matrix.



(b) 256KB-WSS μB.

Figure 6: Histograms showing the percentage improvement in the ACET of two benchmark programs provided by sharing the LLC allocation, instead of providing isolated partitions. The x-axis gives different precentages, and for a given percentage, the histogram shows the number of cases observed across all considered LLC-allocation sizes that exhibited that percentage improvement. Negative (resp. positive) percentages indicate that isolation (resp. sharing) is better.

schedulability experiments the variant of our general allocation strategy mentioned at the end of Sec. 3 in which the Level-C/OS LLC area is partitioned instead of shared.

Obs. 3. LLC isolation had a greater impact on WCETs as LLC space approached ICAS. Beyond this point, WCETs were only marginally greater than in an idle system, implying that unmanaged hardware resources (TLB, memory bus, memory controllers—see Sec. 3) had only a small impact.

In Fig. 5(a), the WCET with LLC isolation becomes quite close to that in an ideal system at four colors, which yields an LLC area matching the μB task's WSS, and therefore its ICAS. A similar trend can be seen in Fig. 5(c) at ten ways.

Obs. 4. Isolation with respect to both the LLC and DRAM banks improved WCETs over LLC isolation alone especially when the allocated LLC area is less than ICAS.

This effect can be seen in insets (a) and (c) of Fig. 5. Note that, if the allocated LLC area is at least the given task's ICAS, then DRAM bank isolation has only a small impact.

Because Level-C PETs are based upon ACETs, we care about ACETs as well, even for provisioning Levels A and B (refer to our earlier discussion of MC analysis in Sec. 2).

Obs. 5. The WCET trends noted above also apply to ACETs. ACETs were lower than WCETs by approximately 5-10% (resp., 80%) for the μB task (resp., Matrix program).

This can be seen by comparing the left-side insets to the right-side insets in Fig. 5. (The 5-10% reduction in the case of insets (a) and (b) may be somewhat hard to see because of the scale.) The Matrix program exhibits a relatively lower ACET because it is less deterministic than the μB task.

Impact of sharing at Level C. If hardware isolation is provided, then ACETs can be computed without much regard for any background workload. However, the situation is murkier for Level-C tasks assuming the default configuration in Fig. 4 since they share the same LLC area and DRAM banks. To better understand this issue, we conducted experiments in which a mixture of DIS programs was executed at Level C and ACETs were determined for individual programs. The following observation follows from these experiments.

Obs. 6. Level-C ACETs for Level-C tasks increased when the allocated Level-C LLC area was reduced or when the utilization of the background Level-C workload was increased.

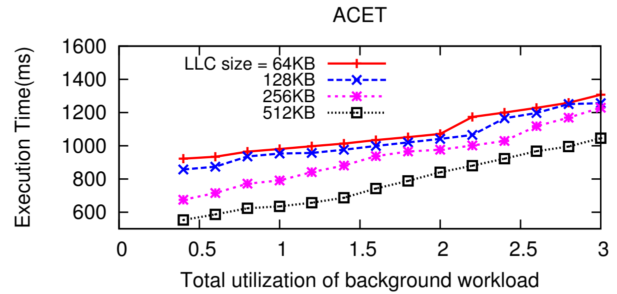


Figure 7: ACETs of the Matrix program with varying LLC sizes and background workloads.

Data supporting this observation is given in Fig. 7, which gives ACETs for the Matrix program assuming various LLC area sizes and total background Level-C utilization. To determine Level-C PETs for Level-C tasks in practice, some domain knowledge would be required when defining an appropriate background workload. In our schedulability experiments, we determined such PETs by "indexing into" a graph similar to that in Fig. 7, which is reflective of the assumption that the background workload is a mix of DIS programs.[6] In Sec. 4.3, we present the results of runtime experiments involving observed timeliness that partially validate this and other provisioning assumptions made in this paper.

OS isolation. Our new MC[2] prototype isolates the OS from Levels A and B with respect to the LLC and DRAM banks (recall Fig. 4). We examined the impact of this feature by conducting an experiment in which a Level-B μB task was executed with and without OS isolation. The μB task was modified to invoke a dummy system call once per loop iteration that allocated and read 16 pages of memory. While such a system call may seem somewhat extreme, the point here is that if OS isolation is not provided, then predictability can be lost, unless the code paths the OS will take are known with high assurance. Fig. 8 shows measured execution times for 100 jobs of the μB task, with and without OS isolation.

[6] This method must actually be iteratively applied because a Level-C tasks's ACET depends on the total average-case utilization of all other Level-C tasks, and hence their ACETs. However, given the scale of our schedulability experiments, which involved millions of task systems, this was infeasible. As a result, we applied this method in a more conservative manner that actually penalized MC[2] with isolation (see [18] for details).

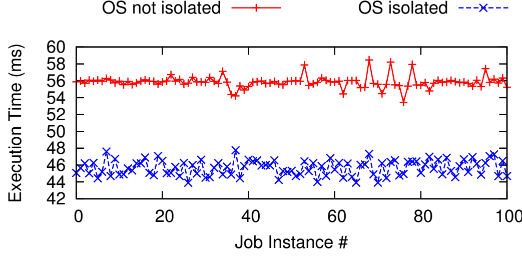


Figure 8: The effect of OS isolation.

Obs. 7. OS isolation substantially reduced the WCET and ACET of the μB task.

OS-related overheads can induce much pessimism in schedulability analysis [4], but Fig. 8 and additional data available online [18] suggest that per-overhead costs can be significantly reduced through OS isolation. One potential concern, however, is that restricting the OS to execute within a smaller LLC area might increase its own execution times unacceptably. However, in additional experiments, we found that providing OS isolation increased system-call overheads by a mere $35\,\mathrm{ns}$ in the worst case and by $15\,\mathrm{ns}$ on average.

4.2 Overhead-Aware Schedulability Experiments

To quantify the gains afforded by criticality-cognizant isolation, we randomly generated millions of task systems and evaluated their schedulability with implementation-related overheads considered under the following scheduling- and resource-allocation schemes.

- **MC^2-ISO**: MC^2 with DRAM-bank and LLC isolation at Levels A and B (as depicted in Fig. 4).
- **MC^2-FULL-ISO**: MC^2 with DRAM-bank and LLC isolation at Levels A and B, and LLC isolation at Level C (where Level-C LLC isolation is achieved via creating per-core LLC areas via way-based partitioning, as discussed at the end of Sec. 3). Differences in schedulability between MC^2-ISO and MC^2-FULL-ISO allow us to analyze the tradeoffs noted in Obs. 2 at Level C.
- **MC^2**: MC^2 with no DRAM-bank or LLC isolation. This scheme provides the advantage of MC analysis only.
- **PEDF-ISO**: Partitioned EDF (PEDF) with DRAM-bank and LLC isolation. This scheme provides the advantage of hardware-management only, thus all tasks use Level-A PETs. PEDF has been shown in previous work [4] to be perhaps the most competitive known HRT scheduling algorithm for multiprocessors when considering implementation overheads.
- **PEDF**: PEDF with no DRAM-bank or LLC isolation.
- **EDF**: EDF on only one core. As stated in Sec. 1, this scheme represents the current industry best practice of disabling all but one core.

These six schemes allow us to independently investigate the gains afforded by isolation and MC analysis, and fully quantify the value of combining both approaches.

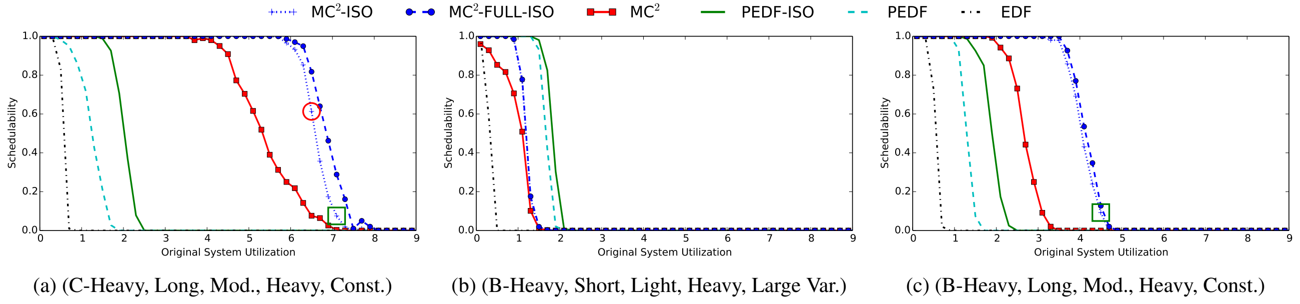
	Choice	Level A	Level B	Level C
Criticality Utilization Ratios	A-Heavy	[50, 70)	[10, 30)	[10, 30)
	B-Heavy	[10, 30)	[50, 70)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	AB-Moderate	[35, 45)	[35, 45)	[10, 30)
	AC-Moderate	[35, 45)	[10, 30)	[35, 45)
	BC-Moderate	[10, 30)	[35, 45)	[35, 45)
	All-Moderate	[35, 45)	[35, 45)	[35, 45)
Period (ms)	Short	{3,6}	{6,12}	[3,33)
	Contrasting	{3,6}	{96,192}	[10,100)
	Long	{48,96}	{96,192}	[50,500)
Task Util.	Light	[0.001,0.03)	[0.001,0.05)	[0.001,0.1)
	Moderate	[0.02,0.1)	[0.05,0.2)	[0.1,0.4)
	Heavy	[0.1,0.3)	[0.2,0.4)	[0.4,0.6)
Max Reload Time	Light	[0.01, 0.1)	[0.01, 0.1)	[0.01, 0.1)
	Moderate	[0.1, 0.25)	[0.1, 0.25)	[0.1, 0.25)
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
Level-A Inflation (%)	Constant	[0.5, 0.5)	[0.5, 0.5)	[0.5, 0.5)
	Small Variation	[0.3, 0.7)	[0.3, 0.7)	[0.3, 0.7)
	Large Variation	[0.1, 0.9)	[0.1, 0.9)	[0.1, 0.9)

Table 2: Task-set parameters and distributions.

Schedulability experimental framework. In our schedulability experiments, we assumed that Level-C and -B PETs are defined to be ACETs and WCETs, respectively, and Level-A PETs are defined by applying an inflation factor to Level-B PETs, as is commonly done in industry [28]. Task sets were randomly generated by using five uniform distributions to choose task and task-set parameters. The specific distributions to use were selected from the per-distribution choices listed in Tbl. 2. These distributions are defined with respect to the single-core EDF scheme. All combinations of these choices were considered. These distributions determine the criticality utilization ratio (*i.e.* the fraction of the overall utilization assigned to each criticality level), task periods, task utilizations, the maximum LLC reload time after a preemption or migration (specified as a fraction of overall task execution time), and per-task Level-A inflation factors (which are similar to those considered by Vestal [28]).

At a high level, our overall experimental framework was as follows. First, the specific five distributions to use were selected from among the choices listed in Tbl. 2. Second, task and task-set parameters were generated under the single-core EDF scheme using these distributions. Third, based on the generated EDF PETs, PETs were generated for other isolation configurations and MC-provisioning assumptions (*e.g.*, PETs should be smaller in schemes that provide isolation compared to those that do not)—this step is described in greater detail in App. B. Fourth, task parameters were adjusted to account for relevant overheads—this step is described in greater detail in App. A. Finally, the resulting task set was checked for schedulability under each considered scheme. In the third and fourth steps, the adjustments to apply were based upon measurement data. (We collected 6GB of task execution-time data and 9GB of overhead data.)

The distributions in Tbl. 2 were defined to enable the systematic study of different factors impacting schedulability, such as MC analysis, isolation, and overheads. We denote each combination of distribution choices using a tuple notation. For example, (C-Heavy, Long, Moderate, Heavy, Constant) denotes using the C-Heavy, Long, Moderate, *etc.*, distribution choices in Tbl. 2. We call such a combination



(a) (C-Heavy, Long, Mod., Heavy, Const.) (b) (B-Heavy, Short, Light, Heavy, Large Var.) (c) (B-Heavy, Long, Mod., Heavy, Const.)

Figure 9: Representative schedulability plots.

a *scenario*. We considered all possible such combinations, and for each one, we generated between 100 and 2,000 task sets, while ensuring that enough were generated to estimate the mean schedulability under a given combination to within ± 0.05 with 95% confidence.

For the schemes that support LLC isolation, we determined allocated LLC areas using our previously proposed optimization method [8], which involves solving a linear program. Under the MC^2-FULL-ISO scheme, we divided the overall Level-C area into fourths (rounding as necessary) to give per-core areas. For schemes requiring task partitioning, we used the worst-fit-decreasing bin-packing heuristic.

Schedulability results. In total, we evaluated the schedulability of approximately three million randomly generated task sets, which took roughly 18 CPU-days of computation. From this abundance of data, we generated over 500 schedulability plots, of which three representative plots are shown in Fig. 9. The full set of plots is available online [18].

Each schedulability plot corresponds to a specific task-set catergory corresponding to a specific combination of the parameter distributions in Tbl. 2. To understand how these plots are interpreted, consider Fig. 9(a). For this plot's task-set category, the circled point indicates that 61% of the generated task sets with EDF utilizations of 6.5 were schedulable under MC^2 with criticality-cognizant isolation. Note that, because the x-axis represents system utilizations under the single-core HRT EDF scheme, it is possible under MC^2 to support systems with an EDF utilization exceeding four, as the MC-provisioning assumptions decrease PETs.

We now state several observations that follow from the full set of collected schedulability data. We illustrate these observations using the data presented in Fig. 9.

Obs. 8. MC provisioning assumptions improved schedulability significantly for approximately 68% of the considered scenarios. Within this 68%, the MC^2 schemes were typically able to schedule at least two to three cores' worth of additional utilization in comparison to the PEDF schemes.

This observation is supported by insets (a) and (c) of Fig. 9. The scenario in inset (a) corresponds to the industry-inspired motivation underlying the specification of MC^2 (see Sec. 2), as in this scenario, only tasks of rather light utilizations exist at Levels A and B. For some scenarios where loads are concentrated at higher criticality levels, the MC^2 schemes also yielded substantial schedulability improvements. Inset (c) provides an example.

Obs. 9. LLC and DRAM-bank isolation improved schedulability only mildly for all scenarios under PEDF. However, such improvements were quite substantial for over 90% of scenarios that benefited from MC^2, enabling MC^2-ISO to schedule one to two cores' worth of additional utilization compared to MC^2 in most cases.

Fig. 9(c) gives one example of this for B-Heavy task sets. In this case, Level-B tasks are afforded the benefits of the LLC while the unmanaged cases are not. However, isolation yields little improvement under PEDF. This is because, under PEDF-ISO, *all* tasks are assigned to cores and contend for LLC allocations, while in MC^2, Level-C tasks share LLC space apart from Level B. This shows that criticality-cognizant isolation can provide major schedulability benefits not seen in criticality-oblivious isolation.

Obs. 10. PEDF outperformed MC^2 in roughly 22% of the considered scenarios, particularly those most affected by overheads (short periods, light utilization tasks, or both). In approximately 10% of the other scenarios, MC^2 and PEDF essentially tied.

Fig. 9(b) presents a scenario for which PEDF outperformed unmanaged MC^2 at certain utilizations, due to additional overheads for MC^2. However, for most scenarios, the benefits of MC provisioning outweighed the disadvantages of additional overheads under MC^2.

Obs. 11. Adding LLC isolation for Level C in MC^2 resulted in differences between MC^2-ISO and MC^2-FULL-ISO that were rather neglible in 60% of the considered scenarios and noticably favored MC^2-FULL-ISO in 10%

In the specific scenarios in Fig. 9, the impact of adding Level-C LLC isolation ranges from negligible (inset (b)) to slight (inset (c)) to moderate (inset (a)).

4.3 Case Studies

The conclusions drawn in our schedulability study are predicated on our provisioning assumptions, in particular the assumption that Level-C tasks can be reasonably provisioned based on measured ACETs. To investigate the validity of this assumption, we created ten task systems for the scenarios in insets (a) and (c) of Fig. 9 corresponding to the highest-utilization point in the MC^2-ISO curves with non-zero schedulability, as highlighted by a square in each figure. (The scenarios of these two insets represent interesting

use-case categories for MC^2-ISO.) Each task system was composed of synthetic μB tasks for Levels A and B, and DIS benchmarks for Level C. Each of these task systems was executed for ten minutes each. Across all of these task systems, there were no Level-A or -B deadlines misses. At Level-C, there were deadline misses—recall this is acceptable as Level-C is SRT—but of the Level-C tasks, the largest relative deadline miss (response time divided by period) was 1.35, and the largest deadline-miss ratio was 1.40%. These results are likely acceptable for most SRT applications. These results suggest that our provisioning assumptions are reasonable, and support our analytical schedulability results.

5 Conclusion

We presented a significant extension to the MC^2 framework that provides LLC and DRAM-bank isolation and that isolates the OS from high-criticality tasks. We also presented the results of extensive experiments (with substantially more data found online [18]) in which the impact of the newly provided isolation mechanisms was assessed individually as well as collectively from a system-wide schedulability point of view. To our knowledge, this is the first work to explore criticality-cognizant hardware-management techniques with the goal of improving platform utilization, the first work that considers isolating the OS from application-level real-time tasks, and the first work to show that the one-out-of-m problem can be effectively addressed through criticality-cognizant hardware management.

This paper suggests many avenues for future work. While we have considered a vast array of possible LLC and DRAM-bank configurations, we made several assumptions to keep the design space tractable, such as isolating high-criticality tasks in the LLC and DRAM banks. Other allocation strategies could potentially yield improved results. We also assumed that Level-C tasks were provisioned based on ACETs. While our case studies suggest this assumption yields positive results, in the future we plan to explore alternative provisioning assumptions, and the tradeoff between increased platform utilization and deadline tardiness and miss ratios. For example, if we provision Level-C tasks based on the 70$^{\text{th}}$ percentile, how much smaller would observed tardiness be? Regarding our implementation itself, the most pressing concern is further extensions to handle shared pages.

References

[1] A Alhammad, S. Wasly, and R. Pellizzoni. Memory efficient global scheduling of real-time tasks. In *RTAS '15*.

[2] S. Altmeyer, R. Douma, W. Lunniss, and R.I. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS '14*.

[3] N. Audsley. Memory architecture for NoC-based real-time mixed criticality systems. In *WMC '13*.

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

[5] A. Burns and R. Davis. Mixed criticality systems – a review. Technical report, Department of Computer Science, University of York, 2014.

[6] M. Campoy, A. Ivars, and J. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Sys. Workshop '01*.

[7] Certification Authorities Software Team (CAST). Position paper CAST-32: Multi-core processors, May 2014.

[8] M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In *RTSS '15 (to appear)*.

[9] G. Giannopoulou, N. Stoimenov, P. Huang, and L.Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *EMSOFT '13*.

[10] M. Hassan, H. Patel, and R. Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *RTAS '15*.

[11] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *RTAS '12*.

[12] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable cache-aware memory allocator. In *ECRTS '11*.

[13] 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 '14*.

[14] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. on Comp. Sys.*, 10:338–359, 1992.

[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 '15*.

[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 '14*.

[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 '13*.

[18] N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. Full version of this paper, available at `http://www.cs.unc.edu/~anderson/papers.html`.

[19] D. Kirk. SMART (strategic memory allocation for real-time) cache design. In *RTSS '89*.

[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 *WICERT '13*.

[21] Y. Krishnapillai, Z. Wu, and R. Pellizzoni. ROC: A rank-switching, open-row DRAM controller for time-predictable systems. In *ECRTS '14*.

[22] LITMUS$^{\text{RT}}$ Project. http://www.litmus-rt.org/.

[23] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *ICPACT '12*.

[24] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed criticality real-time scheduling for multicore systems. In *ICESS '10*.

[25] J. Musmanno. Data intensive systems (DIS) benchmark performance summary, Aug. 2003.

[26] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity environment. In *ECRTS '14*.

[27] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *DATE '10*.

[28] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS '07*.

[29] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *ECRTS '13*.

[30] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicoore platforms. In *RTAS '14*.

[31] 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 '12*.

A Overhead Accounting

To account for implementation-related overheads in our schedulability experiments, we applied several existing overhead-accounting techniques [4, 8]. While a complete, formal description of these techniques is beyond the scope of this paper, in what follows, we give a high-level description of the techniques employed, and highlight the most relevant ideas. We account for all overhead sources through PET *inflation*, *i.e.*, increasing the PET of each task before evaluating schedulability. We considered the following overhead sources, defined in [4]: cache-related delays, context switching, release latency, timer ticks, scheduling, job release, and inter-processor interrupts (IPIs).

Of these overhead sources, cache-related delays are the most significant. We handled such delays similarly to our previous work [8]. However, in our earlier work, we assumed that if Level-A jobs are split into subjobs when creating a cyclic-executive dispatching table, then PETs were already defined to account for the effects of such splitting. This assumption is reasonable if jobs are split at specific points in the job logic. However, our implementation uses budgets at Level A, and therefore we do not *a priori* know the slicing point. Therefore, we must account for cache-related delays in our overhead model.

Of the remaining overheads considered, we applied techniques pioneered by Brandenburg [4] for PEDF and GEDF, for Level B and Level C, respectively, with minor modifications to account for interactions among criticality levels in MC^2. When analyzing the subsystem for each criticality level, we used measured overheads acquired using similar assumptions as used for PETs, *e.g.*, at Level C, average-case measured overheads were considered. Also, in the case of scheduling and release overheads, we have to account for per-core partitioned scheduling and release overheads at Levels A and B, and also global scheduling and release overheads that may be incurred on any core for Level C. Releases and IPI overheads from task migrations at Level C may cause delays at all criticality levels.

Our MC^2 implementation heavily uses PET budgets. The management of such budgets gives rise to a new overhead source. These overheads are incurred when a budget is replenished or depleted, and are accounted for similarly to other overheads by inflating PETs.

B PET-Generation Process

The PETs assumed in Sec. 4.2 are based on an analytical model, which we derived through distilling the measured execution-time data discussed in Sec. 4. This appendix describes this PET-generation model in greater detail. As described in Sec. 4.1, all PETs required in our schedulability experiments are defined based on EDF-scheme PETs, which correspond to A-inflated WCETs in an idle system with the full LLC allocated to the task in question. We denote this WCET parameter as C_i^0 for task τ_i. In our experimental framework, the C_i^0 values are obtained implicitly from the randomly generated task utilizations and periods. All execution-time values used to obtain all other PETs for τ_i

Exec. Time Val.	WCET or ACET?	Idle or Load?	LLC Iso.?	DRAM Iso.?	LLC Area
C_i^0	A-infl. WCET	Idle	N/A	N/A	Entire LLC
C_i^1	A-infl. WCET	Load	Yes	Yes	Entire LLC
C_i^2	A-infl. WCET	Load	Yes	Yes	Any Area
C_i^3	A-infl. WCET	Load	Yes	No	Any Area
C_i^4	A-infl. WCET	Load	No	No	Any Area
C_i^5	WCET	Load	All Relevant Cases		Any Area
C_i^6	ACET	Load	Yes	Yes	Any Area
C_i^7	ACET	Load	Yes	No	Any Area
C_i^8	ACET	Load	No	No	Any Area

Table 3: Generated PET values.

Task Level	PET Level	MC^2 -ISO	MC^2 -FULL-ISO	MC^2	PEDF -ISO	PEDF	EDF
A	A/HRT	C_i^2	C_i^2	C_i^4	C_i^2	C_i^4	C_i^0
	B	C_i^5	C_i^5	C_i^5	N/A	N/A	N/A
	C	C_i^6	C_i^6	C_i^8	N/A	N/A	N/A
B	A/HRT	N/A	N/A	N/A	C_i^2	C_i^4	C_i^0
	B	C_i^5	C_i^5	C_i^5	N/A	N/A	N/A
	C	C_i^6	C_i^6	C_i^8	N/A	N/A	N/A
C	A/HRT	N/A	N/A	N/A	C_i^2	C_i^4	C_i^0
	B	N/A	N/A	N/A	N/A	N/A	N/A
	C	C_i^8	C_i^7	C_i^8	N/A	N/A	N/A

Table 4: Assignment of execution time parameters to PETs.

for different isolation and analysis assumptions are listed in Tbl. 3. Tbl. 4 shows how these values are used to define all PETs under each schem. The columns of Tbl. 3 indicate how each execution-time value is defined (*i.e.*, whether the value is a Level-A-inflated WCET, a non-inflated WCET, or an ACET, whether the system is assumed to be under load or idle, *etc.*). Each of these values is generated by applying scaling factor(s) to the prior-listed execution-time values. We present an overview of this entire process here.

Step 1: Generate C_i^1 by scaling C_i^0 to account for interfering workload. We choose C_i^1 uniformly from $[120, 150)\%$ of C_i^0, based on WCET measurement data in idle and loaded systems with the full LLC allocation.

Step 2: Generate C_i^2 by scaling C_i^1 for different LLC allocations. Our C_i^0 values are defined from generated utilizations. The process for generating such utilizations was carefully defined to produce trends similar to those seen from measurement data. Since our C_i^1 values are simply scaled versions of our C_i^0 parameters, similar utilization trends will be seen when utilizations are defined in terms of C_i^1 values. Fig. 10 illustrates typical generated utilizations. As seen in this figure, task utilizations monotonically decrease with increasing LLC space and converge at the ICAS. This is in accordance with Obs. 3. To reflect this, we obtain C_i^2 values for different LLC-allocation choices by applying a

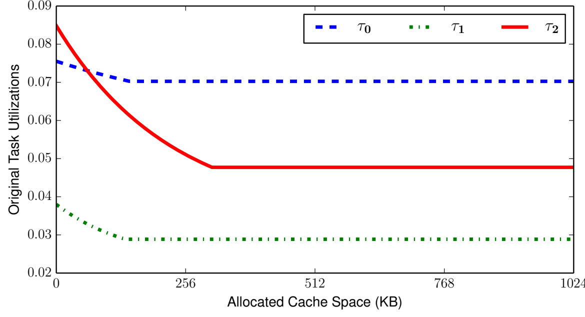


Figure 10: Utilizations generated under different LLC allocations for three example tasks.

scaling factor to C_i^1 that exponentially increases with the minimum of the ICAS and LLC space. The actual scaling factors employed were selected to reflect measurement data.

Task ICASs are deduced using the Load Time parameter in Tbl. 2. The two both hinge on a task's cache footprint. Our Load Time parameter was defined to reflect Obs. 1, which showed that cache isolation can improve a task's WCET by up to 277%. In fact, in data available online [18], the improvement was as much as 400%. For example, when the Light Load Time distribution is assumed, LLC isolation typically reduces WCETs by 20-50%, while when the Heavy distribution is assumed, the reduction is typically 200-500%. In addition, for all parameter combinations, tasks at Levels A and B tend to be more insensitive to LLC space than those at Level C. This reflects the underlying motivation for MC^2 that Level-A and -B tasks will tend to be rather deterministic fly-weight tasks and that Level-C tasks will tend to be more complex (see Sec. 2).

Step 3: Generate C_i^3 by scaling C_i^2 to account for shared DRAM banks. As seen in Fig. 5, the impact of DRAM bank isolation on task execution times tended to range from imperceptible to 20% under small LLC allocations. Based on these results, we uniformly choose C_i^3 to be $[100, 130)$% of C_i^2 to account for the lack of bank isolation. Similar to the last step, this step is affected by the task ICAS and LLC allocation.

Step 4: Generate C_i^4 from C_i^3 based on known worst-case shared-cache behavior. When sharing a cache, cross-core interference may prevent a program from reusing any data in any shared cache blocks, thus eliminating any benefit from the LLC in the worst case. Therefore, we define C_i^4 to equal C_i^3 for the case when the allocated LLC space is zero.

Step 5: Generate all Level-B PETs from previously generated Level-A PETs. Using the A-Inflation Factor in Tbl. 2, all Level-B PETs can be computed from corresponding Level-A PETs. This gives us all C_i^5 values.

Step 6: Generate C_i^6 and C_i^7 to reflect expected ACET:WCET ratios under cache isolation and varying background workloads. ACET:WCET ratio trends will depend on the given background workload, *i.e.*, the total utilization of all competing tasks. Based on WCET:ACET ratio trends observed for benchmark and synthetic programs under different background workload utilizations, we identify an appropriate distribution from which to uniformly choose an

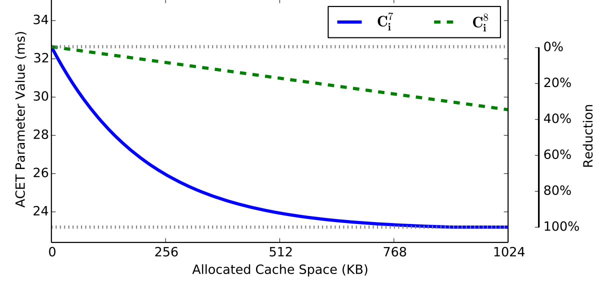


Figure 11: Comparison of C_i^7 and C_i^8 for a generated task

ACET:WCET ratio for each task. For all tasks, these ratios are chosen uniformly among a range of percentages. For Level-C tasks, these ratios range over 20-40% for the lightest background workloads, and over 30-60% for the heaviest. For Level-A and -B tasks, these ratios range over 50-70% for the lightest background workloads, and 80-100% for the heaviest. This reflects our assumption that higher-criticality tasks tend to be more deterministic in their execution than Level-C tasks.

Step 7: Generate C_i^8 to reflect differences between ACETs for a fully unmanaged system and ACETs for a cache-isolated system. From Fig. 5 and Obs. 6, we see that ACETs in an unmanaged cache gradually decline in a linear fashion as the allocated LLC space increases, even beyond the ICAS of the task. However, these ACETS generally remain higher than ACETs under cache isolation. When the LLC allocation is zero, both ACETs are the same, since LLC management does not affect tasks bypassing the LLC. To reflect this behavior, we generate C_i^8 as shown in Fig. 11. On the right axis, we depict a scale showing the range of C_i^7's reduction in value as the allocated LLC space increases. On this scale, C_i^7 is at 0% reduction under zero allocated LLC space, and 100% under maximum allocated LLC space. C_i^8 at maximum allocated LLC space for the Matrix program would fall at approximately 50% on this scale. For each generated task, we choose a value from 30-70% on this scale for our generated C_i^8 at maximum LLC space. At zero allocated LLC space, C_i^8 matches C_i^7. For all other LLC allocation sizes, we interpolate values for C_i^8 linearly between values generated for zero allocated LLC space and maximum allocated LLC space.

From these steps, we now have all required PETs. We note once again that this process produces a *model* for producing PETs. As such, all claims resulting from our schedulability experiments apply only within the context provided by this model. Still, we have taken great pains to ensure that the range of PETs generated by this model encompass those that we have seen based on real measurement data, and that trends among related PETs for the same task correspond to those seen in our measurement data.