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. The key reason is the "one-out-of-m" problem: when checking real-time constraints on a multicore platform with m cores, analysis pessimism can easily negate the processing capacity of the additional $m-1$ cores. Two major approaches have been investigated previously to address this problem: mixed-criticality allocation techniques that seek to provision less-critical software components less pessimistically, and hardware-management techniques that seek to make the underlying platform itself more predictable. In this paper, the results of an experimental investigation are presented that shows that applying both approaches together can have a much greater impact than applying either alone. This investigation is based on a new variant of the MC^2 (mixed-criticality on multicore) framework that enables tasks to be isolated by criticality level with respect to the hardware resources they access.

1 Introduction

Multicore platforms have the potential of enabling a wealth of new computationally intensive features in safety-critical domains such as in the avionic and automotive industries. However, certifying the real-time correctness of a system running on m cores can necessitate using 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.

Two major approaches for addressing this problem have been investigated. The first approach involves leveraging the fact that systems often have tasks of differing criticalities. The goal here is to produce analysis that enables less-critical tasks to be provisioned less pessimistically, even though they must co-exist with more-critical tasks in the same system. The second approach focuses instead on the real root of the problem, namely shared hardware resources, such as caches, buses, and memory banks, that are not predictably managed. By introducing such management, the goal here is to reduce pessimism by enabling tighter task execution-time estimates. While each approach seems promising by itself, a better way forward might be to combine both approaches.

Isolation versus sharing. Such a combination strategy gives rise to new allocation tradeoffs pertaining to sharing and isolation 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. We also present the results of an experimental investigation that demonstrates the virtues of applying hardware-management techniques in a mixed-criticality (MC) setting. Our new platform extends a framework called MC^2 (mixed-criticality on multicore) [11, 23, 27], which has been the subject of continuing research by our group, by adding support for several hardware-management techniques.[2] 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, disruptions caused by the OS have not been considered before in work on hardware management techniques. We regard MC^2 as a rich and interesting platform for our investigation because (as discussed later) it supports several criticality levels (and 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, we explain how we extended MC^2 to provide support for LLC

*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] In fact, the U.S. Federal Aviation Administration is currently considering the possibility of *mandating* such an approach when multicore platforms are used in avionic systems.

[2] MC^2 should not be confused with a similarly named European project that began several years *after* work on MC^2 commenced.

and DRAM bank management and OS isolation. This new MC^2 variant is highly configurable and breaks new ground by allowing sharing and isolation tradeoffs to be studied in a criticality-cognizant way.

Second, we present a variety of experiments concerning such tradeoffs that demonstrate the value of managing hardware in MC systems. In the MC^2 configuration considered in these experiments, 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 based on two case-study task systems. For one of these task systems, applying both MC analysis and hardware management on a quad-core ARM machine resulted in a schedulable system with an average-case utilization of 3.592. In contrast, applying only MC analysis, average-case utilization rose to 4.768, applying only hardware management, it rose to 5.466, and applying neither, it rose to 8.688. Thus, for this system, over-utilization can only be avoided if both techniques are applied together, though applying each by itself has some impact.

We are certainly not the first to investigate MC analysis techniques or approaches for managing shared hardware—we review prior related work on these issues later, to properly position our contributions. However, we are the first (to our knowledge) to provide criticality-cognizant isolation—with respect to both the OS and some of the most problematic sources of shared-hardware interference—within a framework as diverse as MC^2.

Organization. The remainder of this paper, we provide needed background and review prior work (Sec. 2), describe relevant implementation details, (Sec. 3), present our experimental results (Sec. 4), and conclude (Sec. 5).

2 Background

In this section, we present necessary background on real-time systems, MC scheduling generally, the MC^2 framework specifically, techniques for isolating tasks with respect to LLCs and memory banks, and prior related work.

Task model. We consider real-time workloads specified using the implicit-deadline *periodic task model* and we assume familiarity with this model. We specifically consider a task system $\tau = \{\tau_1, \ldots, \tau_n\}$, scheduled on m processors,[3] where task τ_i's *period* and *worst-case execution time* (*WCET*) are denoted T_i and C_i, respectively. (Below, the task model specified here is refined to allow multiple execution-time estimates to be associated with the same task.) We denote the jobs released by T_i as $J_{i,1}, J_{i,2}, \ldots$. We denote the *utilization* of task τ_i as $u_i = C_i/T_i$ and the *total system utilization* of τ as $U_{sum} = \sum_i u_i$. 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 $J_{i,j}$ is released at time $r_{i,j}$, has a deadline at time $d_{i,j}$, and completes execution at time t, then its *response time* is $t - r_{i,j}$ and its *tardiness* is $max\{0, t - d_{i,j}\}$. 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 [26]. He observed that, from the perspective of certifying the real-time requirements of a less-critical task, the execution times assumed of more-critical tasks are needlessly pessimistic. Thus, he proposed that schedulability tests for less-critical tasks be altered to incorporate less-pessimistic execution times for more-critical tasks. More formally, in a system with L criticality levels, each task has a *provisioned execution time* (*PET*)[4] 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 [26], observed worst-case execution times were used to determine PETs for tasks at lower levels, and such times were inflated to determined PETs at higher levels. The task model resulting from Vestal's work has come to be known as the *MC task model*.

MC^2. Vestal's work led to approximately 200 follow-up papers on MC scheduling by a variety of authors (see [5] for an excellent survey). Most prior work on this topic has focused on uniprocessors or has emphasized theoretical issues. MC^2 was the first MC scheduling framework for multiprocessors (to our knowledge) [11, 23, 27]. MC^2 is implemented as a LITMUS$^{\mathrm{RT}}$ [21] plugin. In MC^2, four criticality levels exist, 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.[5] Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each core.[5] 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 hyper-period. Level-C tasks are scheduled via a global earliest-deadline-first (G-EDF) scheduler.[5] Level-A and -B tasks are HRT, while level-C tasks are SRT. Level-D tasks are scheduled with no real-time guarantees (so we do not consider them further). MC^2 is

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

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

[5] A RM (EDF) scheduler can be optionally used at Level A (B). Additionally, any G-EDF-like (GEL) scheduler [8] can be used at Level C. Furthermore, Level-C tasks can be defined according to the sporadic task model. For simplicity, we do not consider these options further herein. Other facets of MC^2, such as slack reallocation, schedulability conditions, and execution-time budgeting are discussed in prior papers [11, 23, 27].

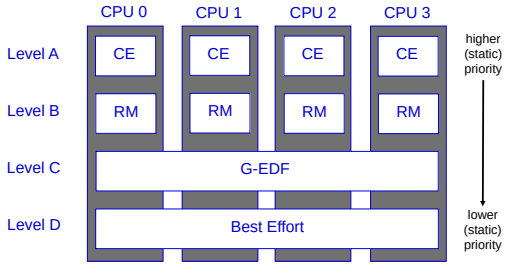


Figure 1: Scheduling in MC^2 on a quad-core machine.

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.

Page coloring. Page coloring is a technique that can be applied to eliminate interference within the LLC and memory banks [14]. We explain the basic idea here with respect to the LLC (which we assume to be set-associative). Consider a physical memory that is subdivided into 4 KB pages and consider each page 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," in which case we reuse color "0," color "1," and so on. Once all pages have been colored, we have the property 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 [18] 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). As we will show, these two techniques can also be used together given support in both hardware and software.

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 and should be avoided if possible. It is possible to prevent a task executing on one processor from causing one executing on a different processor to experience row buffer misses by partitioning DRAM banks among processors [22].

Prior related work. The use of cache partitioning in real-time systems has been investigated before. Kim *et al.* [17] presented a cache-partitioning scheme that allows multiple tasks to share the same cache partition on a single processor (as we do in Sec. 3), but they did not consider MC systems. Altmeyer *et al.* [2] considered uniprocessor scheduling on a system with a direct-mapped cache and examined WCET estimates as a function of cache size. They also presented a cache-partitioning algorithm that is optimal under certain assumptions. 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 [20] and MC [3, 24, 15] systems, on improved timing analysis for MC multicore systems that more accurately assesses memory inference [13], and on making bus accesses more predictable in single-criticality [1, 25] and MC systems [9, 10]. Regarding DRAM-related issues specifically, Kim *et al.* [16] presented detailed analysis for predicting memory access delays in which DRAM characteristics are carefully considered. However, they did not consider MC systems. Yun *et al.* [29] 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.* [19].

Yun *et al.* [28] presented PALLOC, a memory allocator that can specifically allocate pages to provide cache and/or bank isolation. However, that work did not consider MC systems, and their implementation makes use of Linux `CGROUPS`, which are not supported in LITMUS$^{\mathrm{RT}}$. Integrating PALLOC with LITMUS$^{\mathrm{RT}}$ would provide greater memory-allocation flexibility than the approach we present in Sec. 3, but that would be a significant implementation undertaking, and is therefore outside the scope of this work.

Perhaps most closely related to this work is that of Ward *et al.* [27], who also considered cache management in MC^2. In their approach, all potentially accessed pages are prefetched by the OS before job execution, which allows colors to be controlled dynamically by the scheduler. Unfortunately, the pre-fetching logic entails significant overhead and (obviously) can be applied only if it is *a priori* known which pages will be accessed. In our work, we avoid prefetching through static partitioning. That work also only considered a two-criticality variant of the original MC^2, and did not manage code pages, nor manage access to memory banks.

While MC^2 provides four criticality levels, almost all of the work on MC systems cited above focuses on enabling just two. Also, to our knowledge, no comparisons of set- and way-based partitioning have been presented in prior work on real-time multicore systems, nor has cache partitioning

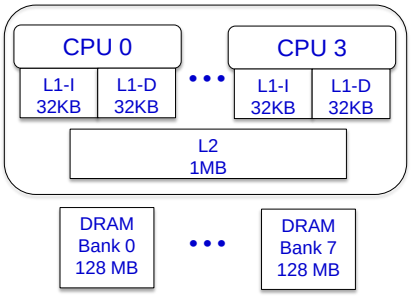


Figure 2: Quad-core ARM Cortex A9.

been considered in MC systems, or in systems in which both partitioned and global schedulers are used. Additionally, we are aware of no actual implementation of a prior resource-allocation framework for MC multicore systems where interference with respect to both the LLC and DRAM memory banks is addressed. Finally, there has been no prior work that addresses interference through shared hardware caused by the OS. Our new MC^2 prototype breaks new ground on all of these fronts. We discuss the design of this prototype next.

3 Implementation

We now describe the hardware-management extensions we added to MC^2. All source code for our new MC^2 implementation is available online [21]. 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 memory references 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. Note that it is possible to combine way- and set-based partitioning to flexibly create LLC areas that can be designated for the sole use of certain tasks.

DRAM banks. Our test platform allows DRAM bank interleaving to be optionally enabled. This option controls how physical memory pages map to banks. 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, if bank interleaving is enabled on our test platform, then the bits within a physical address that determine the mapped-to bank

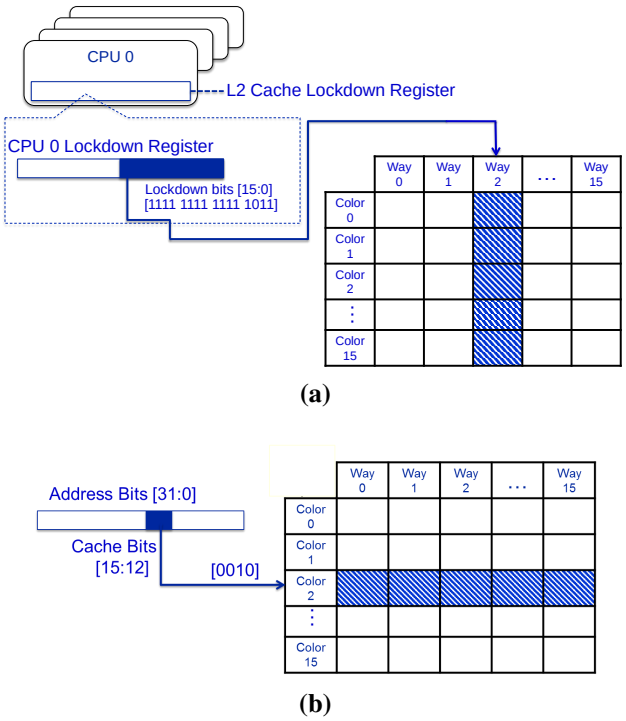


Figure 3: LLC partitioning **(a)** by way and **(b)** by set.

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, to obtain the benefits of bank interleaving. (Under the canonical allocation strategy primarily considered herein, only Level-C tasks access multiple banks.) 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. In our experiments, we were able to fully allocate to these pools all pages from six of the DRAM banks, Bank 2 through Bank 7. As a result, the OS executes entirely within Banks 0 and 1 and our user-level tasks execute entirely within the six other banks. Our page-coloring process is able to color all pages associated with each task, except for a special signal-handling page that should rarely be accessed. However, it is currently limited to allocating only non-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 user-level tasks in the LLC via way-based partitioning. Specifically, whenever kernel code begins executing as the

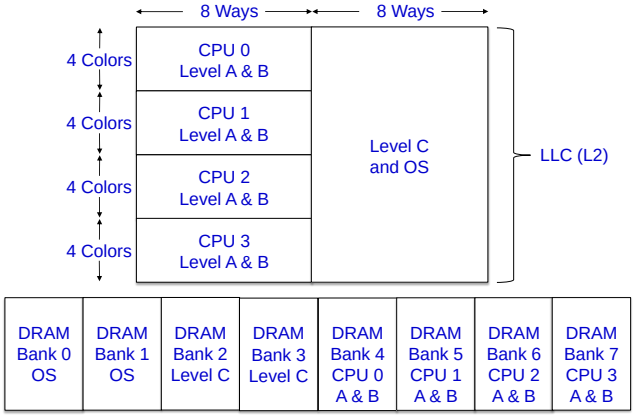


Figure 4: Canonical LLC and DRAM allocation.

result of an interrupt, exception, or system call, we modify the lockdown register of the corresponding CPU so that the OS code can access only a specific set of LLC ways. Together with the DRAM isolation just described, this ensures that the OS can only minimally interfere with user-level tasks.

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 measure-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.

Canonical LLC and DRAM allocation for MC^2. Determining an optimal allocation of LLC areas and DRAM banks to tasks and criticality levels is an interesting optimization problem and is beyond the scope of this paper. However, in a companion paper [7], we show that LLC areas can be optimized for specific task systems by solving a linear program. In this paper, we mainly consider the fixed allocation depicted in Fig. 4. We allow half the LLC to be shared by Level-C tasks and the OS. In prior work on MC^2 [11, 23, 27], 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 rest of the LLC is partitioned among Level-A and -B tasks on a per-CPU basis. That is, the Level-A and -B tasks on a given processor share a partition. This scheme ensures that Level-A and -B tasks do not experience LLC interference due to tasks on other cores (*spatial* isolation). Also, because Level-A tasks have higher priority than Level-B tasks, Level-A tasks do not experience LLC interference due to Level-B tasks on the same core (*temporal* isolation).

As for the DRAM memory, two banks are dedicated to the OS, as described earlier. Additionally, each CPU has a dedicated bank for its Level-A and -B tasks, and the remaining two banks are shared by all Level-C tasks. This scheme ensures strong isolation for higher-criticality tasks, and allows hardware sharing for lower-criticality tasks.

4 Evaluation

In this section, we show how MC allocation and scheduling combined with hardware management as provided in our MC^2 implementation can enable more work to be supported on a multicore platform. We begin in Sec. 4.1 by examining the impacts of LLC and DRAM bank isolation on task execution times. Then, in Sec. 4.2, we examine the impacts of OS isolation. The experiments in both Secs. 4.1 and 4.2 only focus on execution-time data collected for individual tasks. In Sec. 4.3, we turn our attention to supporting sets of tasks. In particular, we consider two case-study task systems and investigate schedulability differences for these task systems under different configurations of MC^2.

4.1 LLC and DRAM Isolation Impacts

In analyzing the impacts of LCC and DRAM isolation, we used synthetic micro-benchmark (μB) tasks because this allowed us to control precisely the LLC- and memory-related characteristics of the tasks under investigation. These μB tasks were specifically designed to create potentially extreme cases to stress the cache and memory subsystems. As such, they can be used 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 reads from a randomly chosen sequence of word addresses that align with the first word in a cache line (32 bytes on our hardware). During each iteration of the loop, every cache line is referenced exactly once. The loop is repeated 500 times, with each loop iteration referencing a different random sequence of cache lines. This 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). Our μB tasks only use data that is statically allocated at program startup.

A critical property of the μB tasks is the assumed *working set* (*WS*), or the set of addresses used to reference data. The assumed *working set size* (*WSS*) is controlled by a single parameter. Note that the average cache reuse distance is the same as WSS because all cache lines are referenced between successive references to the same line (*i.e.*, from one loop iteration to the next).

In our experiments, we collected data for a range of WSSs. However, due to space limitations we mainly limit attention to a WSS of 256KB here. Some additional results are presented for a WSS of 32KB in an appendix, and still further WSSs are considered in an online appendix (available at `http://www.cs.unc.edu/~anderson/papers.html`). A 256KB WS is larger than the L1 data cache on our platform by a factor of eight, but a task's instructions easily fit in the L1 instruction cache. The 32KB WS also fits into the L1 data cache.

We ran experiments in which a μB task was either run alone on one core (we call this the *idle scenario*) or run concurrently with *stress-inducing tasks* running on the other three cores (we call this the *loaded scenario*). The stress-

inducing tasks were configured to use the same random cache-line referencing strategy as our synthetic μB tasks but with a WS expanded to 1MB. We considered the following four isolation configurations.

- **Idle:** The μB task was the only task in the system.
- **Loaded, no cache or memory-bank isolation:** The μB and concurrent stress-inducing tasks accessed the same LLC area and the same 128MB DRAM bank. This case represents the worst-case unmanaged system.
- **Loaded, cache isolation, no bank isolation:** The μB task was isolated in the LLC, but stress-inducing tasks accessed the same DRAM bank.
- **Loaded, cache and bank isolation:** Stress-inducing tasks were executed concurrently with the μB task, but isolation was provided for the μB task in both LLC and DRAM banks.

For each μB task and isolation configuration, we ran experiments with all 256 possible LLC area sizes (given by 1 to 16 ways and 1 to 16 colors) allocated to the μB task. In configurations that isolate the μB task in the LLC, the remainder of the LLC area was used by the stress-inducing tasks. We measured both WCETs and average-case execution times (ACETs) over 100 runs for each scenario.

The data we collected for the 256KB WSS case is shown in Fig. 5; corresponding data for the 32KB WSS case is given in Fig. 10, which is discussed in the appendix. All plots show the μB task execution time (either WCET or ACET) as a function of the number of LLC colors for a number of ways equal 1, 2, 4, 8, and 16. Note that each additional color or way adds 4KB to the total LLC space. For example, 4 colors and 16 ways yields an LLC area of 256KB, which is the μB task WSS considered in Fig. 5.

From these experiments, we make several observations about how WCETs and ACETs are affected by assumptions regarding LLC and DRAM isolation. Our observations about WCETs are highly relevant to Levels A and B of MC^2, as these are HRT levels that would (at least) require a worst-case provisioning. Our observations about ACETs are highly relevant to Level C, since tasks at this level are SRT and thus might be provisioned on an average-case basis.

Obs. 1. LLC isolation reduced WCETs up to 400%.

Consider insets (e) and (g) of Fig. 5. These scenarios differ only in that LLC isolation is provided in the former and not the latter (and DRAM isolation is provided in neither). A 400% reduction in WCET can be noted by examining the data points in these insets for 4 colors and 16 ways. Moreover, by comparing insets (e) and (a), which gives data for an idle system, the WCET for this LLC area-size choice approaches that of an idle system. While a decrease of approximately 400% was the most significant seen, the reduction in other cases is also quite substantial.

Obs. 2. The impact of LLC isolation on WCET becomes more significant as the allocated space approaches WSS.

This can be seen by again examining the data point in inset (e) for 4 colors and 16 ways. The corresponding LLC area size here matches the μB task's WSS, and by adding more colors, the WCET does not substantially improve.

Obs. 3. Under LLC isolation or LLC and bank isolation, allocating of LLC space by adding ways instead of adding colors somewhat improves WCETs.

Recalling again inset (e), the data point for 8 colors and 8 ways gives the same LLC area size as that considered above. While the decrease in WCET is substantial at this point, it is not as much as in the case of 4 colors and 16 ways considered above. A similar trend is seen in inset (c), where isolation is provided for both the LLC and DRAM banks.

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

Data supporting this observation is found in insets (e) and (c). In particular, note where the curves start on the y-axis. The scenarios considered in these two insets differ only in that DRAM bank isolation is provided in the former but not the latter. Note that there is little improvement in WCET from combining bank isolation with LLC isolation when the allocated LLC area is at least the task's WSS. This result is not surprising since isolation in DRAM memory banks reduces memory contention only when data must be fetched as a result of a cache capacity miss.

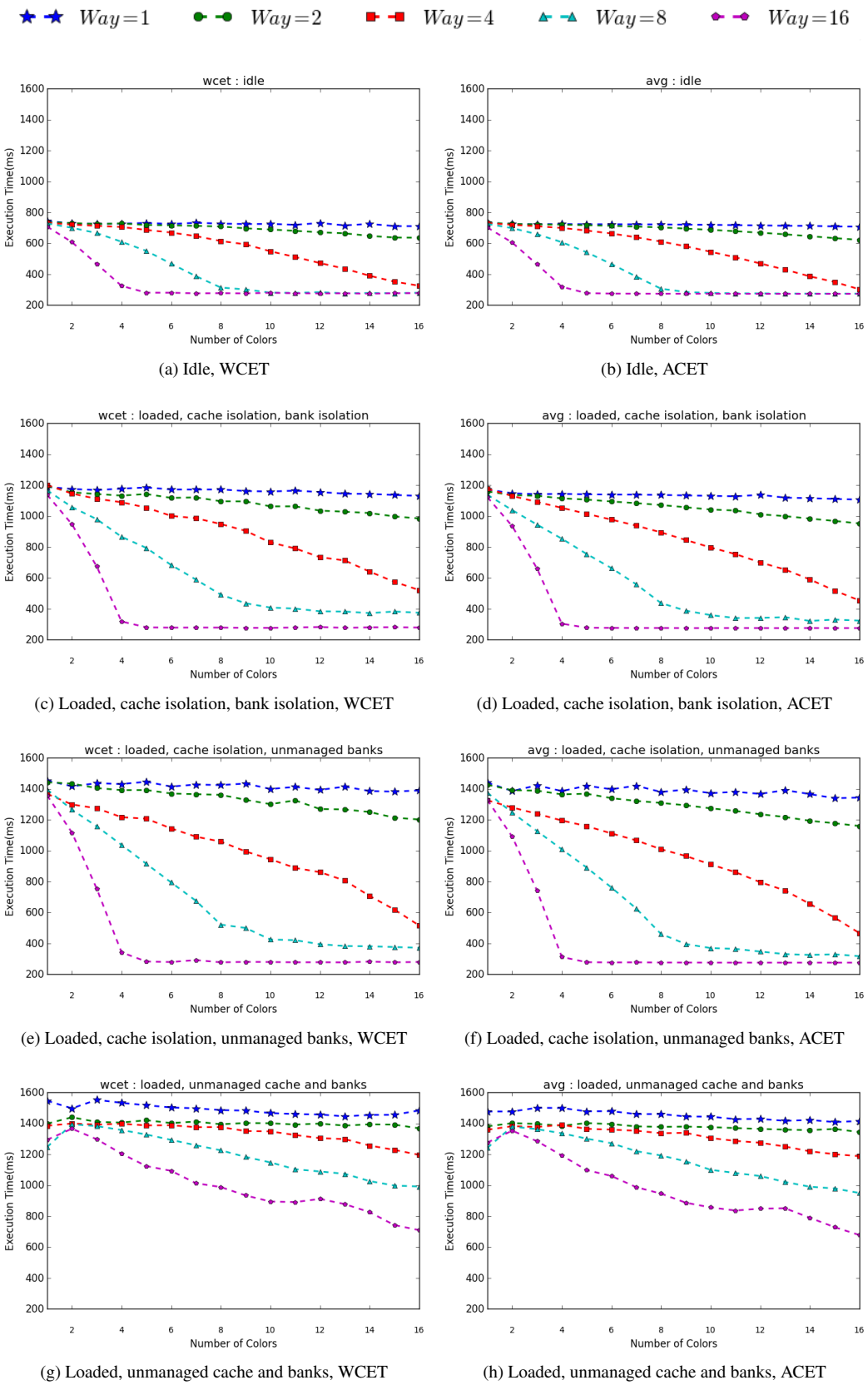
Obs. 5. The effects of isolation on ACETs follow the same improvement trends as for WCETs. Furthermore, ACETs are lower than WCETs by approximately 5-10%.

The insets on the right-hand side of Fig. 5 give measurement data for ACETs. The curves in these insets are quite similar to those for WCETs in the insets on the left-hand side. Although it may be difficult to see given the scale of each plot, ACETs were generally 5-10% lower than WCETs. This rather small difference is due to the very deterministic nature of our μB tasks.

Obs. 6. Assuming Level C in MC^2 is provisioned based on measured ACETs, the ACET data for this *particular* μB task vindicates our Level-C LLC allocation strategy. However, data for other tasks could reveal interesting tradeoffs.

In much of our work on MC^2, we have assumed that Level C is provisioned on an average-case basis. When using a measurement-based process to determine WCETs, it is clearly reasonable to perform measurements in the presence of an intensive cache- and memory-thrashing workload. However, if one is interested in obtaining average-case measurements, then the nature of the background workload that should be considered is somewhat unclear. Insets (b) and (h) of Fig. 5 provide two extremes: in the former scenario, there is no background workload, and in the latter, there is an intensive one. A reasonable measurement process might consider some possibility between these two extremes.

If one were to provision Level C based on the data in inset (b), then one possible conclusion would be that 8 colors and 8 ways would be sufficient because the ACET at this data point



(a) Idle, WCET

(b) Idle, ACET

(c) Loaded, cache isolation, bank isolation, WCET

(d) Loaded, cache isolation, bank isolation, ACET

(e) Loaded, cache isolation, unmanaged banks, WCET

(f) Loaded, cache isolation, unmanaged banks, ACET

(g) Loaded, unmanaged cache and banks, WCET

(h) Loaded, unmanaged cache and banks, ACET

Figure 5: Execution-time data for a WSS of 256KB

is 306ms, which is near the minimum ACET of this μB task. This represents an LLC area that is half the size of 16 colors and 8 ways allocated to Level C in Fig. 4. On the other hand, if one were to provision Level C based on the data in inset (h), then our allocation scheme would give an ACET of 952ms, which corresponds to the data point for 16 colors and 8 ways in this inset. Instead of requiring Level-C tasks to fully share half the LLC, we could instead use way-based partitioning at Level C. In particular, we could divide the Level-C LLC area equally among the four cores, and use lock-down registers to require a Level-C task executing on a given core to use only the two Level-C ways allocated to that core. (This is similar to the way-based OS isolation mechanism described in Sec. 3.) In this case, each Level-C task would execute within an isolated LLC area with 16 colors and 2 ways. How does this way-based alternative compare to that depicted in Fig. 4? For this particular μB task, if we examine inset (f) of Fig. 5, and consider the data point for 16 colors and 2 ways, we see that this scheme would result in a larger ACET of 1159ms. Thus, for this particular task, this way-based allocation scheme would not be a win.

We caution the reader, however, that this conclusion is based on examining one particular μB task. If we were to examine data for tasks with smaller WSSs, or that exhibit less deterministic behaviors, we might reach a much different conclusion. Our intention here is to illustrate that interesting tradeoffs exist when adding hardware management to MC systems. As stated earlier, determining how to best resolve these tradeoffs is beyond the scope of this paper.

This concludes our discussion of the data we collected for the 256KB WSS case. As mentioned earlier, data for the 32KB case is presented in the appendix, particularly in Fig. 10. While we discuss that data in more detail there, we do wish to make one observation concerning it. A quick comparison of the data in Figs. 5 and 10 reveals a striking difference: note the consistently jagged nature of the curves in Fig. 10. At first, we found this behavior quite surprising because a μB task with a 32KB WSS should quickly load all of its referenced memory into the L1 data cache and then experience only hits in the L1 thereafter. However, for reasons explained in the appendix, such an assumption turned out to be faulty. This leads us to one additional observation.

Obs. 7. The presence of the peak WCET and ACET values in Fig. 10 where none was expected provides a cautionary note for designers concerned with provisioning systems.

As explained in the appendix, in all other respects, the data from the 32KB experiments corresponded to expectations. WCETs and ACETs were quite close to idle system execution times, and isolation mechanisms had little effect.

4.2 OS Isolation Impacts

In the experiments discussed so far, no OS system calls were done by any μB task and no attempt was made to isolate the OS from application-level tasks. Here we examine the impact of OS contention and how its effects might be ameliorated using the OS isolation mechanisms proposed in Sec. 3.

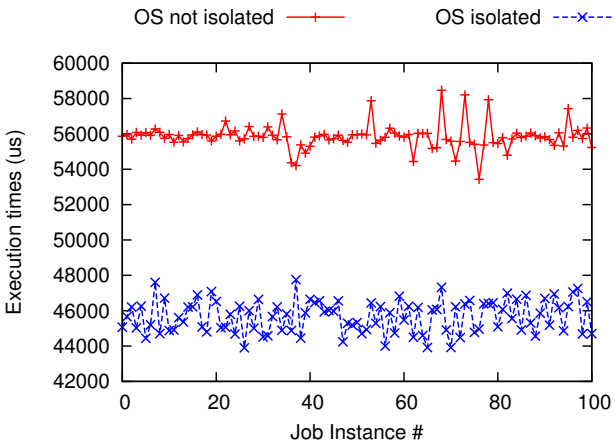


Figure 6: The effect of OS isolation

To assess this, we used the same μB task described earlier and ran it on core 0 at Level B with and without OS isolation. We used the allocation depicted in Fig. 4 to achieve OS isolation. We set the WSS parameter of the μB task to match the size of the cache allocation for the task (a Level-A/B LLC area consisting of 4 colors and 8 ways, or 128KB). To emulate the effects of OS executions, we implemented a dummy system call 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 it is known with high assurance which code paths the OS will take. The system call was inserted in the code for the μB task between iterations through the randomly selected cache lines. As in the prior experiments, we measured execution times over 100 runs for the μB task.

Obs. 8. OS isolation substantially reduced the WCET and ACET of the μB task.

Fig. 6 shows the difference in execution times with and without OS isolation for all 100 experimental runs. The WCET and ACET were both reduced by about 20% when OS isolation was provided. OS-related overheads can induce a great deal of pessimism in schedulability analysis [4], but these experiments suggest that the cost of these overheads can be significantly reduced through OS isolation.

One potential concern with providing OS isolation is that, by restricting the OS to execute within a smaller LLC area, its own execution times might increase unacceptably. To test this, we used the LITMUS$^{\mathrm{RT}}$ API, which provides a user program that measures raw system call overheads [21]. We found that providing OS isolation increased system-call overheads by 35ns in the worst case and by 15ns on average. Such increases are negligible, and are certainly outweighed by the increased execution-time predictability afforded to high-criticality tasks. We also performed additional experiments to explore OS interference that may occur across cores. These experiments are covered in the appendix.

4.3 Case-Study Experiments

The μB experiments presented above show that each individual isolation feature that we added to MC^2 can be applied in a way that lessens task execution times. However, it remains to be seen whether these features can be holistically applied in a way that positively impacts overall schedulability. After all, it could be the case that applying these features in a way that reduces execution times for some tasks increases them for others, with no real gain in terms of schedulability.

To shed light on this issue, we conducted additional experiments involving two case-study task systems. The first system, termed the *LLC-heavy system*, was designed as an example that puts significant pressure on the LLC. This task system consists of multiple instances of nine distinct synthetic tasks. The synthetic tasks used here were constructed similarly to those used in our μB experiments, except that we carefully selected each task's WSS and the number of loop iterations it performed over its WS to achieve desired execution time behaviors.

The PETs for the nine tasks used to define this task system, with and without hardware management, are show in inset (a) of Table 1. We determined PETs through a measurement process (as noted earlier, on multicore platforms adequate static timing analysis tools do not yet exist). Level-C PETs, denoted PET_C, were obtained by measuring average execution times in a loaded and in an idle system, and taking the mean of these two measurements. This is in line with our assumption stated earlier, that Level C, being SRT, might reasonably be provisioned based on average-case execution times. Level-B PETs, denoted PET_B, were obtained by measuring worst-case execution times in a loaded system. Level-A PETs were obtained by inflating Level-B PETs by a factor between 25% to 50%. Such an inflation is in keeping with inflation factors derived from industrial use cases considered by Vestal [26]. The justification for these choices is that both Level B and Level A are HRT, and Level A is of utmost criticality. In the case of hardware management being applied, these measurements were taken with isolation provided as depicted in Fig. 4. In the case of no hardware management, the LLC was disabled and all memory references were directed to the same DRAM bank. All tasks were assigned periods, as indicated in inset (a) of Table 1. As indicated in inset (b), the final task system was constructed by assigning instances of these tasks to criticality levels, and further assigning those assigned to Levels A and B to cores.

The second system, termed the *LLC-light system*, was designed to put significantly less pressure on the LLC. It was defined in a similar manner as the LLC-heavy system, except that the synthetic tasks that were employed have much smaller WSSs. Details are given in Table 2, which is organized similarly to Table 1.

The hardware platform used in this study is the same quad-core ARM platform considered earlier. In both task systems, the total Level-C utilization (*i.e.*, total utilization assessed assuming Level-C PETs) is approximately 15% of the system's capacity for Level A and 20% for Level B, with overall Level-C utilization (across all three levels) being about 90%.

Tasks	PET_A	PET_B	PET_C	Period	WSS, Loop
T0	4/5	3/4	2/3	100	32KB, 50
T1	15/24	11/18	10/13	100	48KB, 50
T2	24/41	18/31	16/22	100	64KB, 50
T3	41/72	31/55	27/38	200	96KB, 50
T4	27/40	20/31	17/21	200	128KB, 20
T5	58/99	44/75	38/54	400	128KB, 50
T6	123/199	93/151	82/109	800	256KB, 50
T7	246/395	186/299	167/221	800	512KB, 50
T8	549/816	416/618	359/459	1600	1024KB, 50

(a) Task PETs with/without hardware management, periods, and WSSs and loop iteration counts.

CPU	Levels	Tasks
CPU0	A	T0, T1
	B	T3, T4
CPU1	A	T2
	B	T4, T5
CPU2	A	T0, T1
	B	T3, T4
CPU3	A	T2
	B	T1, T4, T5
Global	C	4 instances of T6, T7, and T8

(b) Task assignments.

Table 1: LLC-heavy system. All times are in ms.

Tasks	PET_A	PET_B	PET_C	Period	WSS, Loop
T0	12/12	10/9	9/9	100	8KB, 1000
T1	5/5	4/4	4/4	100	16KB, 200
T2	13/13	10/10	9/9	100	16KB, 500
T3	62/63	47/48	46/46	400	8KB, 5000
T4	73/77	55/58	54/55	400	24KB, 2000
T5	186/197	141/149	136/140	800	24KB, 5000
T6	371/393	281/298	271/279	1600	24KB, 10000

(a) Task PETs with/without hardware management, periods, and WSSs and loop iteration counts.

CPU	Levels	Tasks
CPU 0	A	T0, T1
	B	T3
CPU 1	A	T0, T2
	B	T3
CPU 2	A	T0, T1
	B	T4
CPU 3	A	T1, T2
	B	T3
Global	C	6 instances of T5 and T6

(b) Task assignments.

Table 2: LLC-light system. All times are in ms.

This distribution of work across levels is somewhat difficult to justify, but it was motivated by guidance provided to us by several industry practitioners.

Results. We examined the schedulability of these two task systems under four system configurations: **(i)** MC^2 analysis is used with all aforementioned isolation features assumed; **(ii)** MC^2 analysis is used but no isolation features are assumed; **(iii)** MC^2 analysis is not used but all isolation features are assumed; **(iv)** MC^2 analysis is not applied nor is any isolation assumed. Under these four configurations, the total Level-C utilization (across all levels) of the LLC-heavy system was found to be 3.592, 4.768, 5.466, and 8.688, re-

spectively. Moreover, under configuration (i), the system was deemed schedulable using the tests presented in [23]. Schedulability clearly cannot be guaranteed under the other configurations because the system is over-utilized. Referring back to the one-out-of-m problem mentioned in the introduction, this is a good example of a challenging system to support. It significantly over-utilizes the assumed hardware platform if no measures are taken to address this problem, and still over-utilizes it if either MC analysis or hardware management alone is applied.

In contrast, for the LLC-light system, total Level-C utilizations (across all levels) was found to be 3.661, 3.721, 4.229, and 4.395, respectively. Moreover, schedulability under MC^2 can be ensured in both configurations (i) and (ii). This system highlights the fact that the one-out-of-m problem may actually not be that severe for workloads consisting of simple tasks that have small WSSs. On the other hand, even for such workloads, isolation may be desirable. For example, the isolation provided to Level A in MC^2 comes quite close to providing Level-A tasks with the illusion that they run on dedicated uniprocessors. Such a strong isolation guarantee could enable static timing analysis tools for uniprocessor platforms to be adapted for the multicore case.

5 Conclusion

We have presented a significant extension to the MC^2 framework that provides LLC and DRAM bank isolation and that isolates the OS from application-level tasks. We have also presented the results of extensive experiments 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 on applying hardware management techniques in a criticality-cognizant way, particularly within a context as complicated as MC^2, and the first work that considers isolating the OS from application-level real-time tasks.

This paper suggests many avenues for future work. Perhaps most importantly, further work is needed to better understand how to optimize hardware allocations in a crititality-cognizant way. In a companion paper, we have taken some initial steps in this regard by presenting a linear-programming-based scheme for optimizing allocated LLC areas [7]. Regarding our implementation itself, the most pressing concern is further extensions to handle shared pages. Support for dynamically allocated kernel pages is also needed. Integrating ideas from PALLOC may be a promising way forward in this regard [28].

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] M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In submission.

[8] J. Erickson, B. Ward, and J. Anderson. Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *Real-Time Systems*, 50(1):5–47, 2014.

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

[19] O. Kotaba, J. Nowotsch, M. Paulitsch, S. Petters, and H. Theiling. Multicore in real-time systems v temporal isolation challenges due to shared resource. In *WICERT '13*.

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

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

[22] 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*.

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

[24] 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*.

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

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

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

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

[29] 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*.

Appendix: Additional Experiments

A Cross-core OS Interference

In Sec. 4.2, we showed how OS isolation can improve performance and predictability for tasks that issue system calls. Here we study how system calls can cause interference across cores, and demonstrate how OS isolation mitigates such interference. Because the OS can allocate pages from any color, if not managed, it can cause LLC evictions of tasks on any core. To measure this effect, we ran two μB tasks concurrently on different cores. One task, which we call the *caller*, issued dummy system calls after each loop iteration, while the other task, the *victim*, did not. The WCET and ACET of each task are depicted in Fig. 7.

Consistent with our previous measurements, the caller task's WCET and ACET were reduced by 20% as a result of OS isolation. Additionally, the victim task also saw a 20% improvement in both WCET and ACET due to OS isolation. Therefore, cache and memory-bank isolation for userspace tasks alone is not sufficient to ensure isolation; the OS must also be isolated to mitigate cross-core timing interference.

B Micro-Benchmark Experiments with a 32KB WSS

To compare to a large WSS μB task, we also considered a μB task with a WSS of 32KB, which is the size of the L1 cache on our test platform. Results from these 32KB-WSS experiments are shown in Fig. 10.

We originally expected that since this WSS matches the size of the L1, all memory references would hit in the L1 with the exception of compulsory misses. However, the results in Fig. 10 do not support this hypothesis. Instead, some odd-numbered color configurations resulted in increased execution times. The baseline execution time of 20ms corresponds to all references hitting in the L1, so these spikes are due to L1 cache misses. To explain these spikes, we must carefully consider the cache structure.

Our L1 data cache is a 32KB 4-way set-associative cache, which has two colors, which we will denote *A* and *B* to distinguish from the LLC colors. The WSS of the considered μB task is 32KB, so there are eight data pages to be colored. The peaks are observed when the colors of the pages are not equally distributed, *i.e.*, it is possible to have five pages of color A and three pages of color B. For example, given three LLC colors, LLC colors 0 and 2 are mapped to color A and color 1 is mapped to color B. Thus, the eight pages are divided amongst the LLC with three pages of color 0, three pages of color 1, and two pages of colors 2. Thus, as shown in Fig. 8, five pages are mapped to color A and three pages are mapped to color B. Since our L1 cache is only 4-way associative, the five pages compete for four ways, thereby causing evictions. This behavior is not observed for all odd-number color configurations. As shown in Fig. 9, assuming nine colors, there are two cases of color assignment. Inset (a) of Fig. 9 shows that the first page is mapped to color B. In this case, there are four pages of color A (white regions)

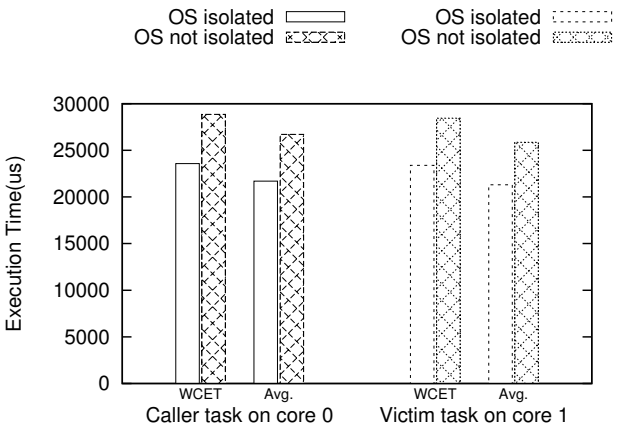
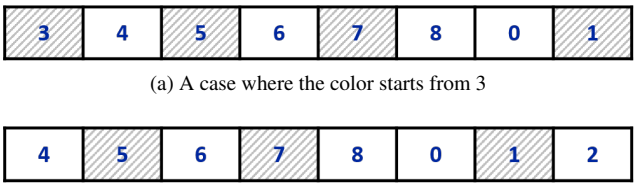


Figure 7: Cross-core OS interference.

and four pages of color B (shaded regions), and L1 evictions do not occur. However, if the first page is mapped to color A, as in inset (b) of Fig. 9, then there are five pages of color A and three pages of color B, which again causes evictions, explaining the execution-time spikes.

As noted in Obs. 7, the presence of these execution-time spikes where none was expected provides a cautionary note for designers concerned with provisioning systems.

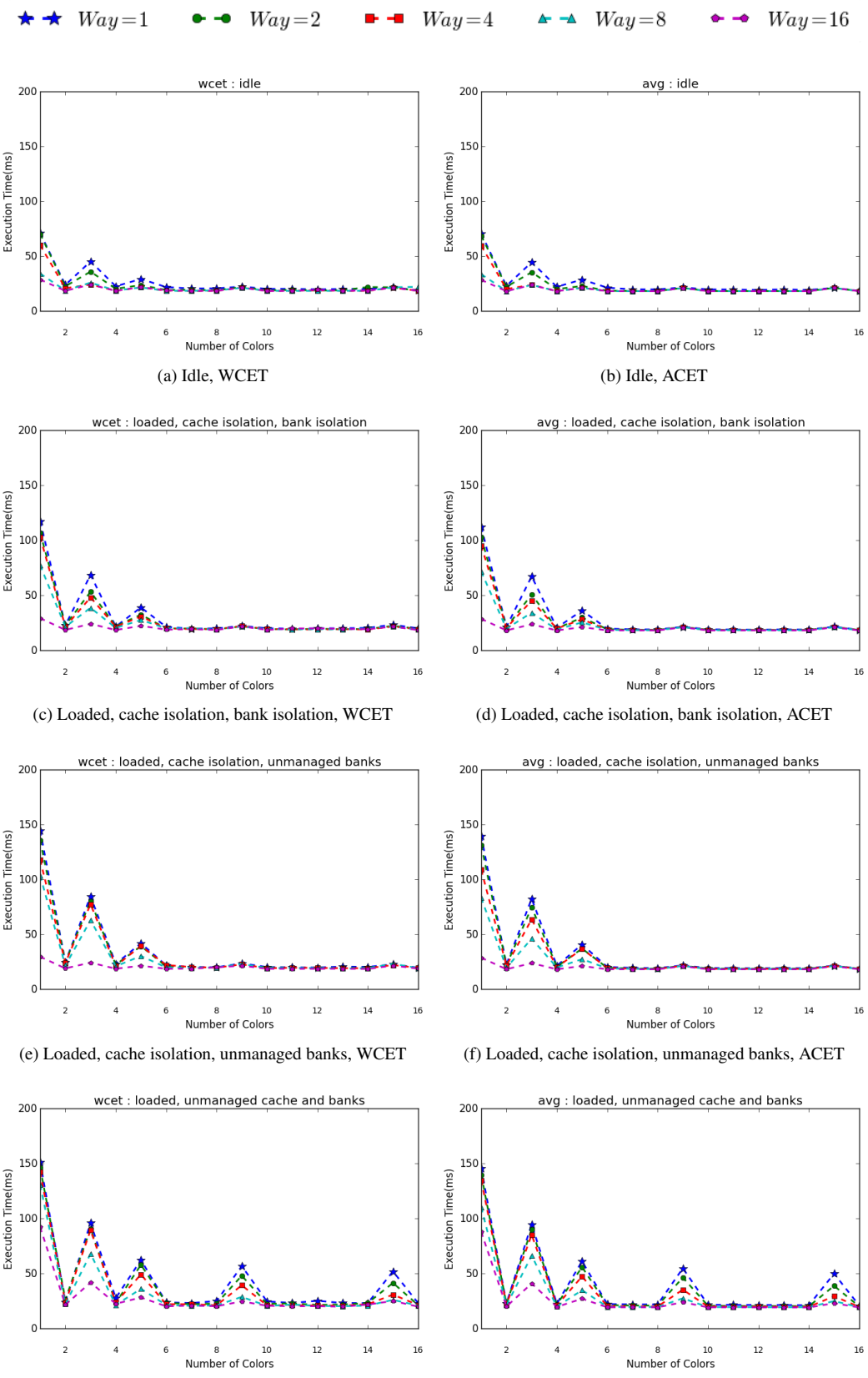
Figure 8: Conflict in the L1 cache.



(a) A case where the color starts from 3

(b) A case where the color starts from 4

Figure 9: Color assignment for eight data pages.



(a) Idle, WCET

(b) Idle, ACET

(c) Loaded, cache isolation, bank isolation, WCET

(d) Loaded, cache isolation, bank isolation, ACET

(e) Loaded, cache isolation, unmanaged banks, WCET

(f) Loaded, cache isolation, unmanaged banks, ACET

(g) Loaded, unmanaged cache and banks, WCET

(h) Loaded, unmanaged cache and banks, ACET

Figure 10: Execution-time data for a WSS of 32KB