Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems [*]

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

Abstract

Recent work involving a mixed-criticality framework called MC^2 has shown that, by combining hardware-management techniques and criticality-aware task provisioning, capacity loss can be significantly reduced when supporting real-time workloads on multicore platforms. However, as in most other prior research on multicore hardware management, tasks were assumed in that work to not share data. Data sharing is problematic in the context of hardware management because it can violate the isolation properties hardware-management techniques seek to ensure. Clearly, for research on such techniques to have any practical impact, data sharing must be permitted. Towards this goal, this paper presents a new version of MC^2 that permits tasks to share data within and across criticality levels through shared memory. Several techniques are presented for mitigating capacity loss due to data sharing. The effectiveness of these techniques is demonstrated by means of a large-scale, overhead-aware schedulability study driven by micro-benchmark data.

1 Introduction

In work on real-time multicore computing, much effort has been directed at a problem that has been termed the "one-out-of-m" problem [12, 22]: certifying the real-time correctness of a system running on m cores can require such pessimistic analysis, 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, the 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.

Much of the pessimism underlying the one-out-of-m problem is attributable to shared hardware resources, such as caches, buses, and memory banks, that are not predictably managed. As such, much of the prior work directed at this problem has focused on enabling tighter task execution-time estimates by providing such management. Additionally, recent work by our group [10, 22] has shown that greater reductions in pessimism are possible when hardware management is coupled with *mixed-criticality* (*MC*) analysis assumptions, as originally proposed by Vestal [34]. Under such analysis assumptions, less-critical tasks are provisioned somewhat optimistically, allowing for increased platform utilization.

While major strides by various research groups have been made in attacking the one-out-of-m problem (see Sec. 6), the ability to support real-world workloads has not yet been realized. A key reason is a lack of support for data sharing among tasks. In prior work, researchers have nearly universally chosen to disallow data sharing because it directly breaks the isolation techniques that underlie hardware management. This choice is certainly reasonable, as a fundamental understanding of the basics of providing isolation is necessary before delving into other complicating factors.

In this paper, we consider for the first time tradeoffs involving data sharing on multicore platforms wherein capacity loss is reduced via the usage of both hardware-management techniques and MC provisioning assumptions. We study such tradeoffs in the context of a pre-existing mixed-criticality framework called MC^2 (mixed-criticality on multicore) [10, 16, 22, 29, 35], which was the focus of our group's prior work noted above. Before describing our contributions, we first provide a brief overview of MC^2.

Hardware management under MC^2. In the MC^2 variant considered herein, three criticality levels exist, denoted A (highest) through C (lowest). (These levels are illustrated in Fig. 1 along with a fourth level.) Level-A and -B tasks have hard real-time (HRT) constraints and are scheduled via partitioning, with per-core scheduling being time-triggered for Level A and priority-based for Level B. Level-C tasks have soft real-time (SRT) constraints and are globally scheduled. In recent work [22], which is built on here, hardware management was introduced with respect to DRAM memory banks and the last-level cache (LLC). This management gives rise to numerous tradeoffs regarding the allocation of LLC areas to groups of tasks and/or criticality levels. In other recent work [10], also built on here, an optimization framework was introduced for sizing these LLC areas [10].

Contributions. The hardware management recently added to MC^2 relies on the ability to provide strong *isolation guarantees* to higher-criticality tasks with respect to DRAM banks and the LLC. The introduction of data sharing can *break any illusion of isolation*. In this paper, we examine the

[*]Work supported by NSF grants CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

[1]Multicore-related certification difficulties are extensively discussed in a recent position paper from the U.S. Federal Aviation Administration [8].

adverse impacts caused by data sharing and present methods for lessening them. Our specific contributions are threefold.

First, after providing needed background (Sec. 2), we describe a new implementation of MC^2 that extends the prior one by allowing tasks to communicate through shared memory (Sec. 3). This new implementation allows producer/consumer buffers to be shared within a criticality level and wait-free buffers to be shared across criticality levels. We consider three options for lessening the impacts of data sharing: locking a shared buffer into the LLC, bypassing the LLC entirely when accessing a buffer, and assigning the tasks that share a buffer to the same core (if they are Level-A or -B tasks) so that concurrent sharing is eliminated.

Second, we explain how to modify the pre-existing optimization framework for sizing LLC areas to account for these buffer-sharing options (Sec. 4). We also add to this framework a task-to-core partitioning heuristic (for Levels A and B) that takes buffer-sharing costs into account.

Third, based on previous micro-benchmark data involving task-execution characteristics [22] and newly collected data involving buffer sharing (Sec. 3.3), we report on the results of a large-scale overhead-aware schedulability study in which data-sharing impacts were assessed (Sec. 5). Across all scenarios considered in this study, our techniques reclaimed 84% of the schedulability lost due to unmanaged data when compared to an ideal scheme that has an infinitely large LLC that can store all shared buffers.

Some limited prior work on MC systems exists in which hardware management was applied (see Sec. 6). However, this paper and three earlier papers on MC^2 [10, 22, 35] are the only ones known to us that consider hardware management 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 material.

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* by $\sum_i u_i$. 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 HRT task, and should be bounded by a (reasonably small) constant for any job of a SRT task.

Mixed-criticality scheduling. The roots of most recent work on MC scheduling can be traced to a seminal paper by Vestal [34]. For systems where tasks of differing criticalities exist, he proposed adopting less-pessimistic execution-time assumptions when considering 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 exceed Level-ℓ' PETs. For example, in the systems considered by Vestal [34], observed WCETs were used to determine PETs for tasks at lower levels, and such times were inflated to determine PETs at higher levels.

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

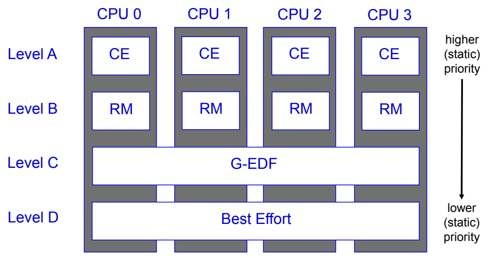


Figure 1: Scheduling in MC^2 on a quad-core machine.

MC^2. Vestal's work led to a significant body of follow-up work (see [7] for an excellent survey). Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors (to our knowledge) [29]. MC^2 was originally designed in consultation with colleagues in the avionics industry to reflect the needs of systems of interest to them. It is implemented as a LITMUS$^{\mathrm{RT}}$ [25] 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 have harmonic periods and commence execution at time 0 (this requirement can be relaxed slightly [29]). 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. In this work, we assume that Level D is not present.

MC^2 with hardware management. In recent work [22], a new MC^2 implementation was developed that provides techniques for managing the LLC and DRAM memory banks. We briefly describe these techniques here. Our description is

[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 [16, 29, 35].

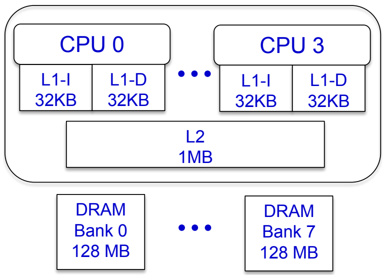


Figure 2: Quad-core ARM Cortex A9.

with respect to the multicore machine shown in Fig. 2, which is the hardware platform assumed throughout this paper. This machine is a quad-core ARM Cortex A9 platform. 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. The LLC write policy is write-back with write-allocate. 1GB of off-chip DRAM is available, and this memory is partitioned into eight 128MB banks.

In the MC^2 variant that provides hardware management, rectangular areas of the LLC can be assigned to certain groups of tasks. This is done by using *page coloring* to allocate certain subsequences of sets (*i.e.*, rows) of the LLC to such a task group, and hardware support in the form of per-core *lockdown registers* to assign certain ways (*i.e.*, columns) of the LLC to the group. (Please see [22] for more detailed descriptions of these LLC allocation mechanisms.) Additionally, by controlling the memory pages assigned to each task, certain DRAM banks can be assigned for the exclusive use of a specified group of tasks. The OS can also be constrained to access only certain LLC areas and/or DRAM banks.

Fig. 3 depicts the main allocation strategy for the LLC and DRAM banks considered in [22]. This strategy ensures strong isolation guarantees for higher-criticality tasks, while allowing for fairly permissive hardware sharing for lower-criticality tasks. DRAM allocations are depicted at the bottom of the figure, and LLC allocations at the top. As seen, Level C and the OS share a subsequence of the available LLC ways and all LLC colors. (On the considered platform, each color corresponds to 128 cache sets.) Level-C tasks (being SRT) are assumed to be provisioned on an average-case basis. 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 that can be determined on a

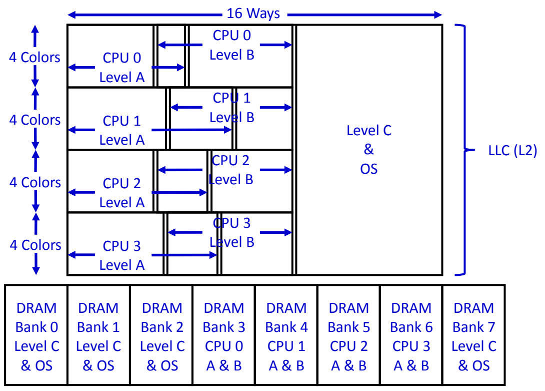


Figure 3: LLC and DRAM allocation. LLC boundaries indicated by double lines are configurable parameters. Note that the Level-A and -B LLC areas for each core can overlap.

per-task-set basis using optimization techniques based on linear programming presented in a prior paper [10]. These optimization techniques seek to minimize a task set's Level-C utilization while ensuring schedulability at all criticality levels.

The MC^2 implementation just described does not provide management for L1 caches, translation lookaside buffers (TLBs), memory controllers, memory buses, or cache-related registers that can be a source of contention [33]. 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 methods for determining PETs are often used in practice.

Problems caused by data sharing. In this paper, we augment the MC task model described above to allow any two tasks to communicate via buffers stored in shared memory page(s) accessible to them both. This modification explicitly *breaks* the isolation properties provided by the pre-existing MC^2 implementation. For example, if two Level-B tasks assigned to different cores share a buffer, then the memory accesses of at least one of them cannot be entirely constrained to its assigned core's DRAM bank. Also, memory accesses by one task can cause the other to experience LLC evictions.

To illustrate this lack of isolation, we conducted an experiment similar to those reported in [22] in which the WCET of a synthetic task with a working-set size (WSS) of 256KB was recorded under the pre-existing MC^2 implementation when executed either in an otherwise-*idle* system or in a *loaded* system that includes a stress-inducing background workload. We considered three loaded scenarios: no hardware management but also no data sharing, and hardware management both without and with data sharing. In the last scenario, each stress-inducing task was configured so that 12.5% of its memory accesses were to shared data. The synthetic task was allocated an LLC area consisting of eight ways and

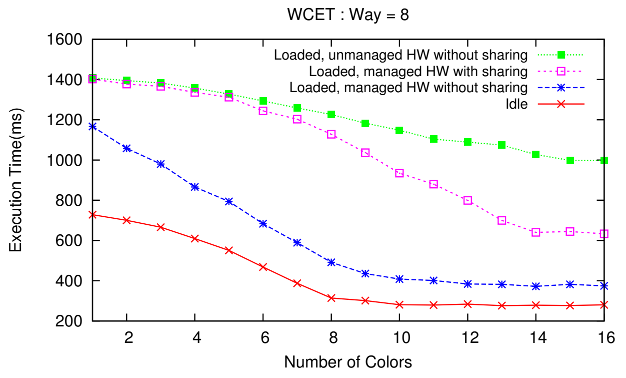


Figure 4: Measured WCET for the 256KB-WSS synthetic task considered in [22] assuming it is allocated eight LLC ways.

some number of colors. The obtained data, plotted in Fig. 4, shows that with at least eight allocated colors and no data sharing, hardware management enabled a WCET near that of an idle system. However, the introduction of sharing caused deterioration close to that of an unmanaged system.

3 Implementation

Our goal in the rest of this paper is to devise methods that eliminate or limit the deterioration just demonstrated. Before discussing such methods, we first describe the inter-process communication (IPC) mechanisms we added to MC^2.

IPC. Linux-based OSs provide several IPC mechanisms, including pipes, named pipes (FIFOs), message queues, and shared memory. Pipes, named pipes, and message queues require data to be copied from user space into kernel space, and then back from kernel space into user space. In contrast, with shared memory, physical memory pages are mapped into the address space of multiple tasks. This allows for lower-overhead IPC, as tasks may read and write buffers stored in shared pages without copying data into and out of kernel space. For this reason, we focus exclusively on shared-memory-based IPC mechanisms in this paper.

We implemented two such mechanisms: *producer/consumer buffers* (*PCBs*) and *wait-free buffers* (*WFBs*). We use PCBs for data sharing among tasks of the same criticality and WFBs for data sharing among tasks of different criticalities. We assume wait-free sharing across levels so that multi-level blocking dependencies (which would greatly complicate MC schedulability analysis) do not occur. Each PCB or WFB is assumed to be written by one task and read by one task. For producer/consumer sharing, buffers must be replicated so that data can be produced without overwriting. For wait-free sharing, overwriting semantics is assumed, but buffer replication is still needed to ensure that read and write operations can occur without interfering with each other [4].

3.1 Mitigating Interference Due to Shared Memory

As discussed in Sec. 2, the introduction of data sharing can cause LLC and DRAM-bank interference. In this paper, we propose to ameliorate such interference by applying the following three techniques in the sequence specified:

Selective LLC ByPass with C-DRAM Assignment (SBP): (i) Designate each buffer accessed by a Level-A or -B task as uncacheable and allocate it from the Level-C DRAM banks. (Note that such a buffer could be shared with a task at Level C.) This eliminates *unpredictable* LLC interference at Levels A and B, at the expense of higher average-case buffer access times due to the lack of caching. **(ii)** Designate each buffer shared exclusively by Level-C tasks as cacheable and allocate it from the Level-C DRAM banks.

Concurrency Elimination (CE): If a buffer is shared by two tasks at Levels A and/or B, then assign both tasks to the same core, and assign the buffer to that core's designated DRAM bank and designate it as cacheable. (Since these techniques are being applied in sequence, this could "undo" a prior designation as uncacheable.) We call such a buffer a *core-local buffer*. This technique eliminates *concurrent* interference with respect to the considered buffer because a core may only execute one task at a time. If a buffer is not core-local, we call it a *cross-core buffer*. Note that all buffers accessed by Level-C tasks are cross-core buffers.

LLC Locking (CL): Permanently lock a buffer in the LLC to eliminate any DRAM-bank contention[5] or unintended LLC evictions. (As explained later, *portions* of a buffer actually can be locked.) This effectively reduces the LLC size for caching code and local data, exposing an interesting tradeoff that is explored in Sec. 5.

CL can be supported by using the lockdown registers mentioned in Sec. 2, using methods proposed by Mancuso *et al.* [28], which entail prefetching the to-be-locked buffer into a way that is then permanently locked by all per-core lockdown registers. Note that locked cache lines may still be read and written from any core, just not evicted.

While it would be desirable to allow a cross-core buffer accessed by a Level-A or -B task to be *dynamically* cacheable, supporting this functionality using lockdown registers is not straightforward. For example, consider a buffer that is shared between Levels A and C. If that buffer were to be cached due to a reference by a Level-C task τ_i^C, then it would be cached in the Level-C LLC area, and thus could be potentially evicted by another Level-C task running on another core. If that were to happen, followed by an immediate reference by a Level-A task, then the buffer would be cached in the Level-A LLC ways. This creates a situation wherein τ_i^C could potentially interfere with Level-A tasks by accessing Level-A ways. To avoid such situations, cross-core buffers accessed by Level-A or -B tasks are either permanently locked in the LLC or never cached in our framework.

Buffer copying rules. Given the semantics of the considered WFBs [4], read (resp., write) operations must copy data to (resp., from) such a buffer, *i.e.*, the data in the buffer cannot be accessed in place. A PCB can be accessed in place

[5]Locked buffers are never evicted, so the write-back policy of our LLC prevents any DRAM-bank contention for locked buffers.

(without copying) if it is core-local, if it is entirely locked in the LLC,[6] or if it is only shared by Level-C tasks. All other PCBs must be copied. These copying rules support a major thesis underlying our implementation that requires the Level-A/B DRAM banks to be kept interference free.

3.2 Kernel-Level Implementation in MC^2

We added a user-level interface to MC^2 where, via a character device, a task is able to request and map shared memory pages into its virtual address space. In our implementation, `mmap()` is used to specify the number of shared pages to map and their permissions, and `ioctl()` is used to specify the DRAM bank and LLC colors of the mapped pages. This implementation extends the page-coloring mechanisms of the prior MC^2 implementation [22].

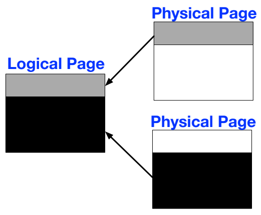


Figure 5: Physical-to-logical page mapping.

We support CL by locking *logical* pages into the LLC, where each such page occupies a single way. A logical page is defined by a set of *physical* pages of the same color. Note that same-colored physical pages will map to the same cache sets. We introduce the concept of logical pages because CL is actually applied at the granularity of *buffers*, not pages. To avoid unnecessarily wasting LLC space, which is a very constrained shared resource, we allow several such buffers to be stored in the same logical page. These buffers can even be allocated by different tasks. Clearly, storing buffers allocated by different tasks in the same physical page would be an egregious protection violation. To avoid this, we store them in different same-colored physical pages with appropriate offsets so that when they are mapped into the LLC, they map to disjoint cache sets.[7] The set of such pages is viewed as a single logical page. This concept is illustrated in Fig. 5. Our implementation is actually more general than just described as it allows *portions* of a single buffer to be split across several logical pages.

3.3 Micro-Benchmarks

To compare SBP, CE, and CL, we ran micro-benchmark experiments in which a shared buffer was either read or written by a measured task. For SBP, we considered only the more-interesting option (i), *i.e.*, the considered page is designated as uncacheable. Under CE, we assumed the buffer was not locked in the LLC. To compare these techniques, we collected 10,000 samples for each choice of *buffer size* (2^NKB, $0 \leq N \leq 8$), *access type* (read or write), and *reference sequence* (random or sequential). For the purpose of measurement only, the measured task was instrumented to access the shared buffer in kernel space via a system call so that the `ktime_get()` API could be used to accurately measure execution times for small buffers.[8] Measurements were obtained in the presence of stress-inducing background tasks. Since we are interested in systems in which the isolation techniques in [22] are employed, the background tasks were constrained to interfere only with the measured task's access of the shared buffer. The allowed interference depended on the technique being evaluated: the background tasks were configured to stress **(i)** the LLC and the specific DRAM bank used by the measured task under SBP; **(ii)** the LLC and the DRAM banks not used by the measured task under CE; and **(iii)** the non-locked LLC ways and the specific DRAM bank used by the measured task under CL. Fig. 6 depicts recorded worst-case times for random writes. Other results, which show similar trends, can be found in an online appendix.

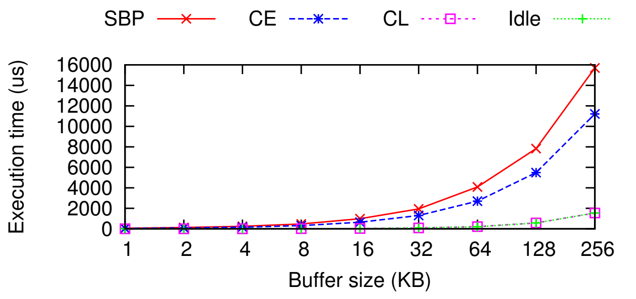


Figure 6: Measured worst-case times for randomly writing a buffer.

Obs. 1. Worst-case buffer-writing times were the lowest under CL and the highest under SBP. CL writing times were typically 2 to 7% of SBP writing times. CE writing times were typically 50 to 60% of SBP writing times.

CL writing times were generally 2 to 3% of SBP writing times for buffers that fit into the L1 cache, and nearer to 7% for other buffers. These results are expected, and are attributable to how the different techniques leverage resources within the memory hierarchy. Memory references will be satisfied from DRAM under SBP, from the LLC under CL, and from some combination of the two under CE, since a miss in the LLC causes a line of eight words to be cached.

4 Optimizations

The micro-benchmark results just described show that CL is clearly the technique of choice if it can be applied to a particular shared buffer. However, there is limited LLC space, so from a system-wide perspective, tradeoffs exist with respect to how the three techniques SBP, CE, and CL are applied. In this section, we show how such tradeoffs

[6]As subsequently discussed, buffers may be partially locked in the LLC. A partially locked cross-core PCB shared with Levels A or B must be entirely copied, but part will be copied from the LLC.

[7]Note that, with this implementation, if a misbehaving task accesses an address in a shared page that is external to the shared buffer it should be accessing (*e.g.*, via a buffer overflow), then this could cause a *temporal* fault for another task by evicting data in a cache line with the same color as the locked logical shared page. However, such a temporal fault cannot compromise logical correctness.

[8]In our new MC^2 implementation, tasks do not actually access buffers in kernel space. Such accesses were performed in kernel space in this experiment to enable more precise measurements.

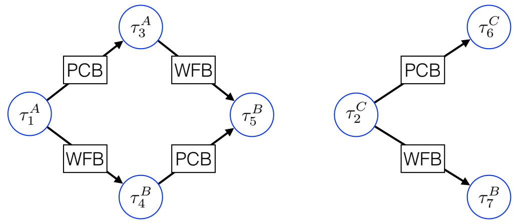


Figure 7: Example task system with producer/consumer buffers (PCBs) and wait-free buffers (WFBs) with tasks at different criticality levels (denoted by superscripts).

can be resolved. We begin by covering necessary additional background. Then, we describe a task-partitioning heuristic that resolves choices related to CE, and modifications to the pre-existing cache-allocation optimization framework [10] that resolves choices related to CL.

Modeling buffers. As discussed in Sec. 2, we consider a set of implicit-deadline periodic tasks $\tau = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$, split across Levels A–C in MC^2. Each task τ_i has a period, and three PETs (one per criticality level).

To represent buffer sharing, we introduce a directed dependency graph, as illustrated in Fig. 7. Each node represents a task, and each edge denotes a shared buffer and is directed from the (single) writer/producer of that buffer to the (single) reader/consumer. Each shared buffer is further specified by a *message size* (the amount of data read or written in one access), a *buffer size* (the message size times the number of message slots in the buffer—recall the discussion about buffer replication in Sec. 3), and its *type* (wait-free for cross-criticality sharing, and producer/consumer for same-criticality sharing). The introduction of producer/consumer buffers introduces precedence constraints (wait-free buffers do not—a read of a wait-free buffer simply obtains the most recently written value, whatever that value is). Because producer/consumer sharing occurs only within a criticality level, we have precedence constraints only within such a level. There is a considerable body of prior work on dealing with precedence constraints, and it is usually assumed that the graph induced by such constraints is a directed acyclic graph (DAG). We assume that here. We also assume that all tasks in the same DAG have the same period.

Schedulability. Prior work has shown that tardiness bounds can be computed for a periodic or sporadic SRT task set with precedence constraints by converting to an "equivalent" independent task set that is then analyzed [27]. Task utilizations are unaltered by this conversion. Since bounded tardiness is ensured at Level C by using utilization-based schedulability conditions [29], Level-C precedence constraints can thus be supported with the same schedulability conditions as before.

We handle precedence constraints at higher criticality levels by introducing release offsets to the task model that determine when a task initially commences execution. For example, in Fig. 7, assuming the common period of τ_1^A and τ_3^A is 100 time units, τ_1^A and τ_3^A could be required to release their first jobs at times 0 and 100, respectively. The introduction of offsets does not impact the utilization of tasks, but can increase the end-to-end response time for a sequence of dependent jobs. We leave detailed end-to-end response-time analysis at Levels A and B to future work, and assume that any bounds that naturally follow from the use of offsets are acceptable provided individual tasks are schedulable. Since Level-A and -B conditions for checking task schedulability are utilization constraints [29], these assumptions imply that we can also assess Level-A and -B schedulability with the same schedulability conditions as before.

With these assumptions, buffer sharing can impact schedulability only by increasing task execution times due to costs incurred in accessing buffers. We discuss this impact next assuming SBP, CE, and CL are applied. Note that, if shared buffers are not managed using any of our techniques, then execution times should be conservatively determined as in an unmanaged system, as the data depicted in Fig. 4 suggests.

Execution-time impacts. As seen in Fig. 1, tasks of higher criticality have priority over those of lower criticality in MC^2. Thus, while each task technically has a PET at each of Levels A through C, only the following PETs are needed for MC^2 schedulability analysis: Level-C PETs for tasks at all levels, Level-B PETs for tasks at Levels A and B, and Level-A PETs for tasks at Level A. In keeping with prior work [22], we assume that Level-C PETs are measured average-case execution times, Level-B PETs are measured worst-case execution times, and Level-A PETs are obtained by applying an inflation factor for safety to Level-B PETs.

To enable the needed execution data to be obtained, we extended measurement-based methods used in our prior work [22] to deal with buffer sharing. We assume that each job consists of three phases. If a job copies a buffer it accesses, then this occurs in an initial *read phase*. The task then executes within an *execution phase* wherein only local data and isolated shared data are accessed. Finally, a *write phase* occurs, if buffers are being copied. Note that, according to the copying rules specified in Sec. 3.1, the read or write phases can be null. If either phase is non-null, then its execution cost can be determined via a measurement process. For safety, this process should include a background workload that stresses the DRAM bank where the non-isolated data being copied to or from is stored.

The buffer copying rules ensure that the duration of a task's execution phase can be determined in the same manner as in our earlier work [22], with one exception: applying CL can cause some data to be permanently locked in the LLC. It may seem surprising that marking a buffer as uncacheable by applying SBP has no impact. However, such a buffer must be a cross-core buffer shared by at least one task at Level A or B. (A buffer shared only at Level C is deemed cacheable by SBP, as is one shared at Levels A and B that is made core-local by CE.) The copying rules require a task to make a local copy of such a buffer before its execution phase.

As for the exception caused by CL, data locked in the LLC is guaranteed to be cache warm at all times, which can decrease the length of a task's execution phase. However, the actual benefits of locked data on a task's execution time are difficult to quantify because they depend on memory-access patterns. Therefore, we conservatively assess these benefits by only considering the time required to load such data once into the LLC. Specifically, we subtract this time from the execution time of a task's execution phase for any locked shared data that the task accesses.

Partitioning heuristics. To support CE, we devised a two-step method that attempts to assign Level-A and -B tasks to cores. In the first step, a modified version of a greedy assignment heuristic proposed by Liu and Anderson [26] is used that attempts to reduce schedulability-related data-sharing costs. The second step is applied only if the first step fails and attempts to find an assignment using the worst-fit decreasing heuristic, which more evenly balances utilization across cores, but is oblivious to data sharing. At a high level, Liu and Anderson's heuristic involves ranking shared buffers by utilization, where a buffer's utilization is based on the time to access it and the period of the accessing task. Modifications to this heuristic were required for our purposes because we have to take into account both Level-A and -B tasks and Level-A and -B PETs when applying the heuristic. Further details can be found in Appendix B.

Prior optimization techniques. Applying CL in a holistic sense requires selecting specific buffers to lock into the LLC. To enable such selections, we modified the prior MC^2 optimization framework [10] briefly mentioned in Sec. 2. That framework was considered previously only in the context of independent task systems. In such a system, a task's PETs vary with the LLC space allocated to it. As a result, relevant utilization values (per-core, per-criticality-level, *etc.*) are dependent on the number of ways allocated to each LLC region in Fig. 3. Under the prior framework, such regions are sized by solving a linear program (LP) that minimizes overall Level-C system utilization (which reduces tardiness for Level-C tasks) subject to meeting all MC^2 schedulability conditions. The LLC-region sizes are determined by variables in the LP that determine, for each region, the number of ways allocated to it. These variables either can be required to be integral, resulting in a mixed integer LP (MILP), or allowed to be continuous and then rounded to integral values, resulting in an ordinary LP. In our prior work, we found that both the MILP and LP versions exhibited similar runtime performance, with the MILP version yielding slightly better results. As such, we assume integer way variables here.

Optimizing buffer locking. We modified the prior MILP to optimize the size and use of the LLC locked-buffer area shown in Fig. 8. Our new MILP determines how much space should be allocated to this area, and how much of each buffer should be locked into it. We allow buffers to be partially

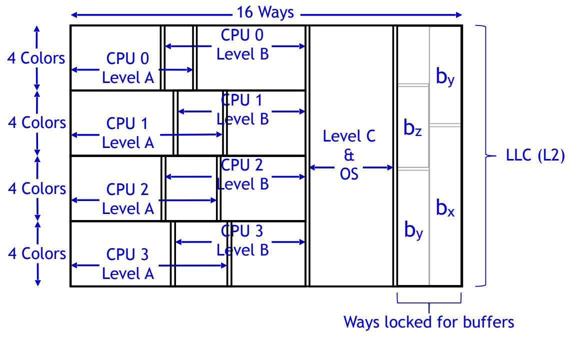


Figure 8: New LLC allocation that extends that in Fig. 3 by allowing several ways to be allocated for holding locked buffers. Several such buffers (b_x, b_y, b_z) are depicted.

locked into the LLC. As a result, continuous variables can be used to represent the amount of each buffer so locked. The only new integer variable needed is one that represents the number of ways allocated to the locked-buffer area. Other aspects of the prior MILP remain unaltered. In applying the prior MILP, we used PETs obtained experimentally. As noted above, these PETs are actually a function of the number of ways allocated to the LLC region a task can use. We found that these functions could easily be dealt with in our MILP by using a piece-wise linear approximation of them [10]. In this work, we have additional execution times that must be considered pertaining to accessing shared buffers. These execution times can also be approximated linearly, as the micro-benchmark data shown in Fig. 6 suggests. Further details can be found in Appendix A.

5 Evaluation

To quantify the benefits of our shared-buffer management techniques, we conducted experiments in which the schedulability of millions of randomly generated task systems was assessed under several scheduling- and resource-management schemes. These schemes, which are summarized in Tbl. 1, are as follows. Under U-EDF, all tasks are scheduled on a single processor using the earliest-deadline-first (EDF) scheduler. This reflects current industry practice for eliminating shared-hardware interference by disabling all but one core. All other schemes use MC^2 scheduling and analysis. Under DSO (data-sharing oblivious), the MC^2 hardware management techniques presented previously [22] are used, but no special techniques are applied for shared buffers. In the next three schemes, the shared-buffer management techniques given in Sec. 3.1 are successively introduced: SBP introduces the SBP technique; SBP+CE then adds the CE technique, with task-to-core assignments being done via the assignment heuristic discussed in Sec. 4; finally, SBP+CE+CL adds the CL technique, where LLC locking decisions are made via the optimization algorithm discussed in Sec. 4. To upper bound the potential gains afforded by the use of our techniques, we also consider Ideal, in which an

Data Category	Benefit	U-EDF	DSO	SBP	SBP+CE	SBP+CE+CL	Ideal
Unshared at Levels A&B	CI	X		X	X	X	X
	BI	X		X	X	X	X
Shared Core-Local	CI	X			X	X	X
	BI	X			X	X	X
	NC	X	X		X	X	X
	CW					locked only	X
Shared w/ AB Cross-Core	CI	N/A				locked only	X
	BI	N/A				locked only	X
	NC	N/A	X			locked only	X
	CW	N/A				locked only	X
Shared at Level C only	CI	X				locked only	X
	BI	X				locked only	X
	CW					locked only	X

Table 1: Benefits accrued under different schemes. Benefits are cache isolation (CI), bank isolation (BI), cache warm (CW) during every access, and no copy (NC) phase required.

LLC of infinite capacity is assumed into which all shared buffers can be locked. (Code and local data are still assumed to be allocated assuming the actual LLC of finite size.)

Task-set generation. The task-set generation process we used extends that used by us previously [22] by accounting for shared buffers. All PETs were defined as discussed in Sec. 4, with the Level-B-to-Level-A inflation factor set to 50% (this is in keeping with results reported by Vestal [34]). Task sets were randomly generated by using six uniform distributions to choose task and task-set parameters. The specific distributions used were selected from the per-distribution choices listed in Tbl. 2. These distributions are defined with respect to the U-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 the maximum percent of a task's working set (WS)—the set of addresses used to reference data—dedicated to reading and writing shared buffers. At a high level, our overall experimental framework refines the following step-wise process used in our prior work [22]:

Step 1 Select six specific distributions from among the distribution categories listed in Tbl. 2.

Step 2 Using the selected distributions from the first four categories, generate task-set parameters under U-EDF.

Step 3 Based on the generated U-EDF PETs, generate PETs for the other schemes in Tbl. 1. This process is informed by micro-benchmark data concerning task execution times, as discussed at length in [22]. In this work, we augment this process by also considering buffer access times, as discussed in Sec. 4.

Step 4 Adjust the generated task parameters to account for relevant overheads. This step is also described in much greater detail in [22]. The actual overhead values applied are based on measurement data.

Step 5 Generate a task dependency graph consisting of a collection of DAGs at each criticality level. The distribution selected from the sixth distribution category in Tbl. 2 determines the number of tasks in each DAG of the graph. Techniques presented by Elliot *et al.* [11] were used to ensure that a wide range of DAG topologies were generated.

Category	Choice	Level A	Level B	Level C
1: 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)
2: Period (ms)	Short	{3, 6}	{6, 12}	[3, 33)
	Contrasting	{3, 6}	{96, 192}	[10, 100)
	Long	{48, 96}	{96, 192}	[50, 500)
3: 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)
4: 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)
5: Max % of WS Shared	Light	[0.001, 0.01)	[0.01, 0.1)	[0.05, 0.1)
	Heavy	[0.15, 0.3)	[0.2, 0.3)	[0.2, 0.7)
6: Tasks Per Graph Component		Task Count (levels not relevant below)		
	Small	{1, 2, 3, 4, 5}		
	Large	{11, 12, 13, 14, 15}		

Table 2: Task-set parameters and distributions.

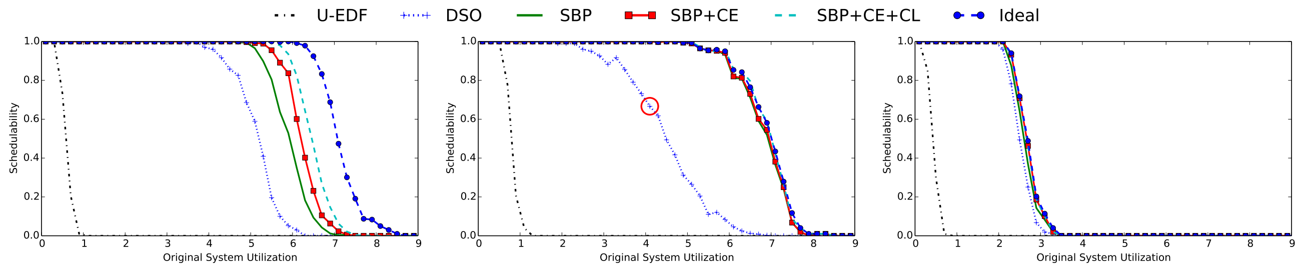
Step 6 Assign wait-free dependencies to cross-criticality task pairs. To keep the parameter space from further exploding, we simply assumed that 1/6 of all dependencies were across criticalities. This reflects the hypothesis that cross-criticality sharing is less common.

Step 7 Generate an upper bound on the fraction of each task's WS that is shared using the distribution selected from the fifth category. We determined actual buffer sizes subject to this upper bound by solving an additional LP that was designed to ensure that the buffer sizes are reasonable given the properties of the tasks that access them.

Step 8 Test the schedulability of the resulting task set under each considered scheme in Tbl. 1.

The distributions in Tbl. 2 were defined to enable the systematic study of different factors impacting schedulability, such as MC analysis, isolation, and shared-buffer sizes. Moreover, these factors were chosen to reflect realistic usage patterns. We denote each combination of distribution choices using a tuple notation. For example, (C-Heavy, Long, Moderate, Heavy, Light, Small) denotes using the C-Heavy, Long, Moderate, *etc.*, distribution choices in Tbl. 2. We call such a combination a *scenario*. We considered all possible such scenarios, and for each utilization in each scenario, we generated enough task sets to estimate mean schedulability to within ± 0.05 with 95% confidence with at least 100 and at most 2,000 task systems.

For schemes that do not lock buffers in the LLC, we determined allocated LLC areas using our prior LP optimization techniques as illustrated in Fig. 3. For schemes not using



(a) (C-Heavy, Long, Mod., Heavy, Heavy, Large) (b) (C-Heavy, Long, Heavy, Mod., Light, Large) (c) (A-Heavy, Contrast, Mod., Light, Heavy, Large)

Figure 9: Representative schedulability plots.

our sharing-aware task-partitioning heuristic, we used the worst-fit-decreasing bin-packing heuristic.

Schedulability results. In total, we evaluated the schedulability of approximately nine million randomly generated task sets, which took roughly 27 CPU-days of computation. From this abundance of data, we generated over 700 schedulability plots, of which three representative plots are shown in Fig. 9. The full set of plots is available online [9].

Each schedulability plot corresponds to a single scenario. To understand how to interpret these plots, consider Fig. 9(b). In this plot, the circled point indicates that 67% of the generated task sets with EDF utilizations of 4.1 were schedulable under the DSO scheme. 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 MC provisioning and hardware management 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. 2. SBP provided moderate schedulability benefits in approximately 60% of the considered scenarios. Moreover, in approximately 30% of cases, SBP provided schedulability near to that of Ideal.

This observation is supported by insets (a) and (b) of Fig. 9. For small-message-size scenarios, as in inset (b), copy-phases have little to no impact on task PETs. As a result, the isolation provided for local data under SBP eliminates the majority of sharing-related schedulability losses without significant schedulability loss due to bypassing the cache or copying. For large-message-size scenarios, as in inset (a), greater copying overheads reduce the benefits of SBP. It is in these scenarios where CE and CL can be most beneficial.

Obs. 3. CE and CL provided mild improvements to schedulability in roughly 20% of considered scenarios.

Fig. 9(a) gives one example of these improvements. In a majority of scenarios, copy-phase lengths are relatively small compared to execution-phase lengths, thus eliminating the need for techniques that primarily improve performance by reducing copying (CE and CL). In such scenarios, the isolation provided by SBP is sufficient to achieve near-Ideal platform utilization.

Obs. 4. In roughly 40% of scenarios, schedulability under all MC^2 schemes, including DSO, was nearly equivalent.

This observation is supported by Fig. 9 (c). In scenarios with light WSs, there is little impact from cache use or shared-data copying. In such scenarios, breaking isolation in the LLC has little effect on task PETs. Similarly, copy-phase lengths have small impacts on PETs. As a result, all MC^2 configurations performed similarly in these scenarios.

Obs. 5. Across all considered scenarios, our combined techniques (SBP+CE+CL) provide 17% better schedulability on average than DSO. Moreover, this represents 84% of the improvement possible when compared to Ideal.

Given the nature of our study, the observations above naturally hinge on our choice of hardware platform. The consideration of other platforms with different caching and memory characteristics might yield different conclusions.

6 Prior Related Work

This work follows a long line of research examining shared-resource contention in real-time systems [23]. Prior efforts have focused on issues such as cache partitioning [3, 17, 21, 36], DRAM controllers [5, 18, 19, 24], and bus-access control [1, 2, 13, 14, 15, 31]. Other work has focused on reducing shared-resource interference when per-core scratch-pad memories are used [32], accurately predicting DRAM access delays [20], throttling lower-criticality tasks' memory accesses [38], and allocating memory [37].

To our knowledge, we are the first to consider in detail the unique impact that data sharing has on hardware isolation under the notion of MC scheduling espoused by Vestal [34], which was proposed with the express intent of *achieving better platform utilization*. Several of the aforementioned papers do target MC systems [5, 13, 14, 15, 18, 19, 24, 30, 38], but only peripherally touch on the issue of achieving better platform utilization, if at all. Also, most of them focus on hardware design. One of these papers [1] considers systems in which tasks share data, but does not consider the specific impact this has on hardware isolation. Hardware isolation under Vestal's notion of MC scheduling is the subject of four prior MC^2-related papers by our group [16, 22, 29, 35]. One of these papers [22] was reviewed in detail in Sec. 2; we refer the reader to [22] for an overview of the other three. In recent work, which was published after we completed our study, data sharing was considered in the context of automotive systems [6].

7 Conclusion

For hardware-management mechanisms to have practical impact, data sharing among tasks must be supported. This need poses a dilemma, as data sharing can directly break isolation properties fundamental to hardware management. In this paper, we considered this dilemma in the context of MC^2, where hardware-isolation mechanisms are used alongside MC provisioning assumptions to further improve platform utilization. In particular, we presented techniques for mitigating this dilemma, and evaluated these techniques via a large-scale, overhead-aware schedulability study, driven by measurement data. This study suggests that our proposed techniques have practical merit. In future work, we plan to augment this study by considering the other IPC mechanisms mentioned in Sec. 3, richer DAG-based task models, and other multicore hardware platforms.

References

[1] A. Alhammad and R. Pellizzoni. Trading cores for memory bandwidth in real-time systems. In *RTAS '16*.

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

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

[4] J. Anderson and P. Holman. Efficient pure-buffer algorithms for real-time systems. In *RTCSA '00*.

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

[6] M. Becker, D. Dasari, B. Nikolić, B. Akesson, V. Nélis, and T. Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *ECRTS '16*.

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

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

[9] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, and F.D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. Full version of this paper, available at `http://www.cs.unc.edu/~anderson/papers.html`.

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

[11] G. Elliott, N. Kim, J. Erickson, C. Liu, and J. Anderson. Minimizing response times of automotive dataflows on multicore. In *RTCSA '14*.

[12] J. Erickson, N. Kim, and J. Anderson. Recovering from overload in multicore mixed-criticality systems. In *IPDPS '15*.

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

[14] M. Hassan and H. Patel. Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In *RTAS '16*.

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

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

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

[18] 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*.

[19] 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*.

[20] 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*.

[21] 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*.

[22] 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. In *RTAS '16*.

[23] 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*.

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

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

[26] C. Liu and J. Anderson. Supporting graph-based real-time applications in distributed systems. In *RTCSA '11*.

[27] C. Liu and J. Anderson. Supporting soft real-time DAG-based systems on multiprocessors with no utilization loss. In *RTCSA '12*.

[28] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *RTAS '13*.

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

[30] 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*.

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

[32] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. Phatak, R. Pellizzoni, and M. Caccamo. A real-time scratchpad-centric OS for multi-core. In *RTAS '16*.

[33] P. Valsan, H. Yun, and F. Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *RTAS '16*.

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

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

[36] M. Xu, S. Mohan, C. Chen, and L. Sha. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS '16*.

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

[38] 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 MILP Example

Due to space constraints, we forego a complete description of the MILP techniques used herein as applied to the ARM Cortex A9 platform. To give readers an understanding of how our MILP techniques work, we present an overhead-unaware version of our MILP for a simpler system than the A9.

Consider a task set $\tau = \{\tau_1, \tau_2\}$ such that τ_1 is a Level-A task and τ_2 is a Level-B task. These tasks share a WFB b with a message size of s_b words, written to by τ_1 and read by τ_2. Both tasks run on a single-core system. Fig. 10 depicts the LLC for this cache, and associated LP variables, discussed later. The LLC contains S^{max} sets, W^{max} ways, and has a cache-line size of one word. τ_1 and τ_2 share a bank, and buffer b is allocated in a Level-C bank. Under this scenario, there is very little competition between criticality levels and cores for cache space. However, this scenario still serves as an exemplar for demonstrating our MILP techniques. Non-negativity constraints are assumed for all LP variables presented in the remainder of this appendix.

Execution-phase modeling. We let $\hat{W}_A$ and $\hat{W}_B$ denote the LP variables for the number of ways allocated to tasks τ_1 and τ_2, respectively. We let $\hat{W}_L$ denote the number of ways used for locked buffer space. We let $\hat{e}_i^A$ and $\hat{e}_i^B$ denote LP variables for the execution-phase time of τ_i under Level-A and -B analysis, respectively. The values of a Level-ℓ execution-phase variable for a given way must be constrained by measured Level-ℓ execution-phase times for the given way allocation. In Fig. 11(a), we show an example plot of what this data might look like for Level-B execution-phase times of τ_2. From the measured execution-phase time data, we can construct linear constraints between adjacent data points, as shown in Fig. 11(b), to ensure that the value $\hat{e}_2^B$ is never less than the measured Level-B execution-phase time for a given way allocation.

Cache-size constraints. To ensure the space allocated to buffers and to tasks does not exceed the size of the cache or produce overlap between the task and buffer LLC areas, our MILP includes the constraint $\hat{W}_A + \hat{W}_B + \hat{W}_L \leq W^{max}$. If we wish to allow overlap in Level-A and -B way allocations, we can replace this constraint with the constraints $\hat{W}_A + \hat{W}_L \leq W^{max}$ and $\hat{W}_B + \hat{W}_L \leq W^{max}$.

Buffer-space constraints. We let $\hat{L}$ denote a continuous LP variable for the number of words in buffer b locked into the cache. For a cache with S^{max} sets, $S^{max} \cdot \hat{W}_L$ cache lines are available for buffer locking. We ensure the buffer data locked into the cache does not exceed the space allocated for buffers using the constraint $\hat{L} \leq S^{max} \cdot \hat{W}_L$. We also constrain $\hat{L}$ based on buffer size. The buffer b requires three slots, so this constraint is $\hat{L} \leq 3 \cdot s_b$

Copy-phase modeling. The cache space allocated to a buffer must be distributed evenly among the message slots of the buffer. For the three-slot WFB b, $\frac{\hat{L}}{3}$ cache lines are read in place from the cache each time a message is read or written. The remaining $s_b - \frac{\hat{L}}{3}$ words of data are copied from a Level-C bank. We let δ_U and δ_L denote the times required for an uncached word and a locked word, respectively, to be read from a Level-C bank, according to Level-B analysis. These times can be determined from measured read/write times for a range of buffer sizes. The Level-B read-phase time r_2^B for τ_2 is the time required to read $s_b - \frac{\hat{L}}{3}$ uncached words from a Level-C bank, and $\frac{\hat{L}}{3}$ cached words from locked cache lines.

$$r_2^B = \delta_L \cdot \frac{\hat{L}}{3} + \delta_U \cdot (s_b - \frac{\hat{L}}{3})$$

Read and write phase times at other criticality levels can be determined in a similar fashion.

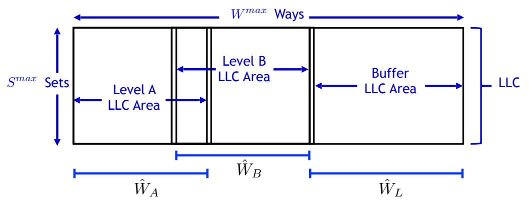
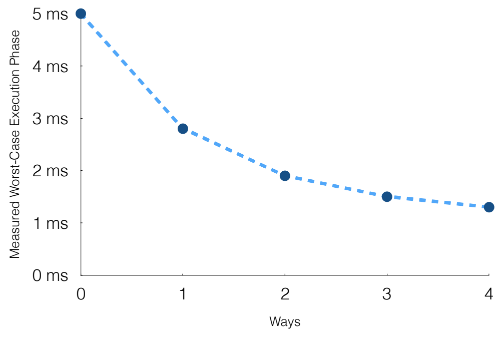
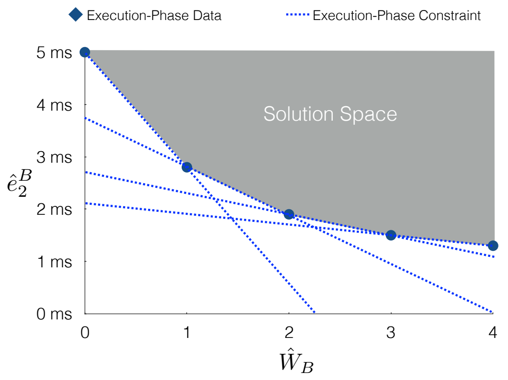


Figure 10: Cache model for MILP description.



(a) Task execution time data for τ_2.



(b) Linear constraints for $\hat{e}_2$.

Figure 11: Construction of linear constraints for τ_2 from task execution time data.

Task utilizations. We let C_i^ℓ and $u_i^\ell = C_i^\ell / T_i$ denote the Level-ℓ PET and utilization, respectively, of task τ_i as expressed in our MILP. Task PETs are expressed in terms of execution-phase variables and copy-phase lengths. For instance, the Level-B PET C_2^B of τ_2 is $\hat{e}_2^B + r_2^B$. Task utilizations are used to express schedulability constraints, all of which are utilization constraints in MC^2.

Schedulability constraints. The system has two utilization constraints, $u_1^A \leq 1$ and $u_1^B + u_2^B \leq 1$, which, in terms of our LP variables, are linear constraints. These schedulability constraints are derived from MC^2 schedulability conditions presented in [29].

Objective function. The MILP as presented does not require an objective function in order to determine values for $\hat{W}_A$, $\hat{W}_B$, $\hat{W}_L$, and $\hat{L}$ that lead to a schedulable system. However, for more complex systems that include Level-C workloads, we include an objective function that minimizes Level-C system utilization in order to improve tardiness bounds at Level C.

This completes our description of MILP techniques applied to a simpler system than the ARM platform. A more complex system under our MC^2 framework requires additional schedulability constraints at each criticality level and on each core, way variables for each LLC area specified in Fig. 8, and additional cache-size constraints. All additional constraints can be formulated in a similar fashion as the constraints presented in this appendix. Task PETs must also factor in overheads. Overheads for our ARM platform can be upper bounded using expressions that are either constant or vary linearly with respect to the values of our LP variables.

B Core-Assignment Heuristics

In Sec. 4, we discussed two steps to our core-assignment method. Here we describe in more detail the first step, a modification to a greedy assignment heuristic proposed by Liu and Anderson [26].

Liu and Anderson's algorithm considers the communication overhead when assigning DAG-based task systems to clusters of processors. In our case, we apply the heuristic to assign tasks at Levels A and B, and for us, each cluster consists of only one core. Greedy choices are made by ranking shared buffers according to the schedulability-related impact of cross-core sharing.

Fig. 12 provides the pseudocode for our modification to Liu and Anderson's algorithm. Algorithm ASSIGN is applied first to all Level-A tasks in the system and then to all Level-B tasks. We use Level-A (respectively, -B) analysis in evaluating core capacities and utilizations when we assign DAGs at Level-A (respectively, -B). Utilizations are evaluated without overheads and assume no ways are allocated to tasks in the LLC.

In Phase 1, we attempt to assign DAGs to cores without splitting DAGs. At line 1, DAGs are ordered based on cross-core sharing "weight." The *sharing weight* for a task is the utilization required for the task to copy from each of its read PCBs once per job and copy to each of its written PCBs once per job assuming all buffers bypass the cache and are allocated in Level-C banks. The sharing weight of a DAG is the sum of the sharing weights of all its tasks. In lines 3-7, we assign DAGs to the first core in P with enough capacity. This is done in worst-fit decreasing order.

In Phase 2, we split up DAGs that were un-assignable in Phase 1. In lines 8-10, tasks in each unassigned DAG are ordered by depth and by sharing weight at same depth. In lines 12-19, we assign tasks in order to cores in worst-fit decreasing order. Tasks are assigned to the first core with enough remaining capacity to hold at least one task. When we cannot assign a task to this core, we disregard the core in future assignments (line 19). This technique is designed to assign as many tasks as possible that likely share buffers to the same core.

```
Γ: A LIST OF DAGS {$\gamma_1, \gamma_2, ..., \gamma_\eta$}
P: A LIST OF CORES {$p_1, p_2, ..., p_m$}
$U(p_i)$: REMAINING CAPACITY OF CORE $p_i$
$u(\gamma_i)$: UTILIZATION OF DAG $\gamma_i$

ASSIGN:
PHASE 1:
 1: Order DAGs by largest sharing weight first
 2: Order cores in $P$ by smallest remaining capacity first
 3: for each $\gamma_i$ in $\Gamma$ in order do
 4:     for each $p_j$ in $P$ in order do
 5:         if $u(\gamma_i) \leq U(p_j)$ then
 6:             Assign all tasks in $\gamma_i$ to $p_j$
 7:             Remove $\gamma_i$ from $\Gamma$
PHASE 2:
 8: for each $\gamma_i$ in $\Gamma$ in order do
 9:     Order tasks in $\gamma_i$ by smallest depth first
10:     Order tasks at same depth by largest sharing
        weight first
11: Order cores in $P$ by smallest remaining capacity first
12: for each $\gamma_i$ in $\Gamma$ in order do
13:     for each task $\tau_j$ in $\gamma_i$ in order do
14:         for each $p_k$ in $P$ in order do
15:             if $u_j \leq U(p_k)$ then
16:                 Assign $\tau_j$ to $p_k$
17:                 Remove $\gamma_i$ from $\Gamma$
18:             else
19:                 Remove $p_k$ from $P$
```

Figure 12: Core assignment algorithm.