Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Tasks*

Namhoon Kim, Stephen Tang, Nathan Otterness, James H. Anderson,
F. Donelson Smith, and Donald E. Porter
Department of Computer Science, University of North Carolina at Chapel Hill
{namhoonk,sytang,otternes,anderson,smithfd,porter}@cs.unc.edu

ABSTRACT

Efforts towards hosting safety-critical, real-time applications on multicore platforms have been stymied by a problem dubbed the "one-out-of-*m*" problem: due to excessive analysis pessimism, the overall capacity of an *m*-core platform can easily be reduced to roughly just one core. The predominant approach for addressing this problem introduces hardware-isolation techniques that ameliorate contention experienced by tasks when accessing shared hardware components, such as DRAM memory or caches. Unfortunately, in work on such techniques, the operating system (OS), which is a key source of potential interference, has been largely ignored. Most real-time OSs do facilitate the use of a coarse-grained partitioning strategy to separate the OS from user-level tasks. However, such a strategy by itself fails to address any data sharing between the OS and tasks, such as when OS services are required for interprocess communication (IPC) or I/O. This paper presents techniques for lessening the impacts of such sharing, specifically in the context of MC^2, a hardware-isolation framework designed for mixed-criticality systems. Additionally, it presents the results from micro-benchmark experiments and a large-scale schedulability study conducted to evaluate the efficacy of the proposed techniques and to elucidate sharing vs. isolation tradeoffs involving the OS. This is the first paper to systematically consider such tradeoffs and consequent impacts of OS-induced sharing on the one-out-of-*m* problem.

CCS CONCEPTS

•**Computer systems organization** → *Multicore architectures; Embedded software; Real-time system architecture;* •**Software and its engineering** → *Memory management; Embedded software; Real-time schedulability; Communications management;*

KEYWORDS

real-time, mixed-criticality, hardware management, multi-core systems, I/O, interprocess communication

*Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, and CNS 1717589, ARO grant W911NF-17-1-0294, and funding from General Motors.

RTNS '18, Chasseneuil-du-Poitou, France

DOI: 10.1145/3273905.3273911

ACM Reference format:
Namhoon Kim, Stephen Tang, Nathan Otterness, James H. Anderson, F. Donelson Smith, and Donald E. Porter. 2018. Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Tasks. In *Proceedings of 26th International Conference on Real-Time Networks and Systems, Chasseneuil-du-Poitou, France, October 10–12, 2018 (RTNS '18),* 11 pages.
DOI: 10.1145/3273905.3273911

1 INTRODUCTION

The desire to host real-time workloads on multicore platforms in safety-critical application domains has been stymied by a problem dubbed the "one-out-of-*m*" problem [12, 30]: when certifying the real-time correctness of a system running on *m* cores, analysis pessimism can be so excessive that 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 problem has led to the common practice of simply disabling all but one core.

The roots of the one-out-of-*m* problem are directly traceable to interference due to contention for shared hardware components: as noted in a recent FAA report [8], interference creates effects that are difficult to predict, and when this happens, analysis pessimism is the inevitable result. Given these roots, the predominant approach for addressing the one-out-of-*m* problem involves affording tasks some degree of hardware isolation, with the ultimate goal of enabling lower (and more predictable) task execution-time estimates [1–4, 10, 11, 13, 15, 16, 18, 23–26, 28, 30–32, 42, 45, 48, 49, 51, 52].

Sharing breaks isolation. In practice, various sources of sharing commonly exist that can break any isolation guarantees afforded to real-time tasks. Such sources include data-sharing among tasks using user-level techniques, read-only sharing through the usage of shared libraries, and the sharing of data between the operating system (OS) and user-level tasks that occurs when tasks invoke OS services for interprocess communication (IPC) or I/O. *No solution to the one-out-of-m problem can be considered complete unless all sources of sharing that exist in a system are addressed.*

In reality, it is generally not possible to address this issue by completely eliminating all interference caused by sharing (unless no task communicates with any other entity!). Thus, when sharing is considered in the context of the one-out-of-*m* problem, the focus inevitably shifts to the weaker goal of lessening its impact. Prior work in this direction by our group has addressed user-level data sharing [10] and the usage of shared libraries [10, 28] in the context of a hardware-isolation framework targeting mixed-criticality systems called MC^2 (mixed-criticality on multicore) [9–11, 17, 28, 30, 36, 48]. However, no work on the one-out-of-*m* problem has heretofore been published that investigates techniques for lessening the impact of OS-induced sharing by optimizing memory allocations. In

this paper, we present the first ever such investigation, which was also undertaken in the context of MC^2.

Contributions. Our investigation required contributions on three major fronts.

First, we extended MC^2 to enable *dynamic memory allocation*. Prior work on MC^2 was constrained to making memory-allocation choices at task creation time. This strategy is insufficient for OS kernel features such as IPC and device I/O, which have complex software stacks that require allocating memory at runtime. While fully static memory allocation may be required for highly critical *hard real-time (HRT)* tasks, MC^2 also supports less-critical *soft real-time (SRT)* tasks that may require greater flexibility in software design. We enabled dynamic memory allocation in real-time tasks by augmenting MC^2's kernel-level memory-allocation functions with controls for requesting specific DRAM or last-level cache (LLC) regions.

Second, we devised options that leverage these controls to allow dynamic memory allocation to be dealt with in an offline MC^2 component that determines DRAM and LLC allocations while optimizing schedulability. Prior to our modifications, this offline component was incapable of determining where dynamic memory allocations should be placed or how tasks should access I/O devices. Our modified offline component continues to optimize schedulability, but can also guide how these finer-grained memory-management controls are used at runtime.

Third, we conducted extensive experiments to evaluate the importance of optimizing data-sharing between devices, the OS, and user-level tasks. These experiments evaluated our modified version of MC^2 in comparison to the preexisting version, which did not consider this type of data sharing. Our experiments included micro-benchmarking efforts and a large-scale overhead-aware schedulability study involving randomly generated task systems. For most of the considered categories of generated task systems, our memory-optimization techniques tended to yield a schedulability improvement of 11% to 14% compared to a naïve allocation, with larger improvements seen for more I/O-intensive categories.

Organization. In the rest of the paper, we provide needed background (Sec. 2), describe how OS-induced sharing introduces interference (Sec. 3) and how we can mitigate such interference (Sec. 4), describe our micro-benchmark experiments (Sec. 5) and schedulability study (Sec. 6), and discuss related work and conclude (Secs. 7–8).

2 BACKGROUND

We consider real-time workloads specified via the implicit-deadline periodic/sporadic task model and assume familiarity with this model. We specifically consider a task system $\tau = \{\tau_1, \ldots, \tau_n\}$, scheduled on m processors,[1] where task τ_i's *period*, *worst-case execution time (WCET)*, and *utilization* are given by T_i, C_i, and $u_i = C_i/T_i$, respectively. If a job of τ_i with a deadline at time d completes at time t, then its *tardiness* is $max\{0, t - d\}$. Tardiness should always be zero for a HRT task, and be bounded by a reasonably small constant for a SRT task.

Mixed-criticality scheduling. For systems with tasks of differing criticalities, Vestal proposed *mixed-criticality (MC)* schedulability

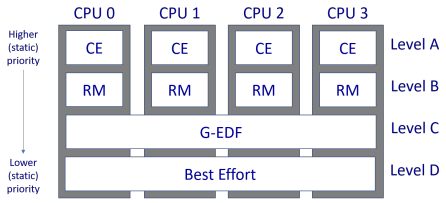


Figure 1: Scheduling in MC^2 on a quad-core machine.

analysis, which uses less-pessimistic execution-time provisioning for less-critical tasks [47]. Under his proposal, if L criticality levels exist, then each task has a *provisioned execution time* (*PET*)[2] specified at each 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 [47], observed WCETs were used to determine lower-level PETs, and such times were inflated to determine higher-level PETs.

MC^2. Vestal's work led to a significant body of follow-up work, surveyed in [7]. Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors [36].

MC^2 is implemented under LITMUS^{RT} [34], an extension of Linux, and supports four criticality levels, denoted A (highest) through D (lowest), as illustrated in Fig. 1. Higher-criticality tasks are statically prioritized over lower-criticality tasks. Level-A tasks are partitioned and scheduled on each core using a time-triggered, table-driven cyclic executive. Level-B tasks are also partitioned but are scheduled using per-core rate-monotonic (RM) schedulers. MC^2 requires the Level-A and -B tasks on each core to be periodic (with implicit deadlines), have harmonic periods, and start execution at time 0. Level-C tasks are sporadic and scheduled via a global earliest-deadline-first (GEDF) scheduler. Level-A and -B tasks are HRT tasks, Level-C tasks are SRT tasks, and Level-D tasks are non-real-time, best-effort tasks. In this paper, we assume that Level D is not present, as it is afforded no real-time guarantees. As in prior work on MC^2 [30], we assume that Level-B and -C PETs are, respectively, WCETs and average-case execution times, and that Level-A PETs are obtained by inflating Level-B PETs by 50%.

Hardware management in MC^2. MC^2 includes several mechanisms for managing the LLC and DRAM banks [30]. We briefly describe how these mechanisms work on the NXP i.MX6 quad-core ARM Cortex A9 evaluation board, which is the hardware platform assumed throughout this paper. Each core on this machine is clocked at 800MHz and has separate 32KB L1 instruction and data caches, as illustrated in Fig. 2. The LLC (the L2 cache) is a shared, unified 1MB 16-way set-associative cache. The system has 1GB of off-chip DRAM memory, partitioned into eight 128MB banks. The system also has one SATA II interface.

MC^2 supports LLC management using per-core *lockdown registers* to assign ways (columns) of the LLC to task groups [30]. It can also allocate sets (rows) of the LLC to task groups for finer partitioning, but we do not explore this option due to space constraints. MC^2

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

[2] Under MC^2, "PET" is used instead of "WCET" because SRT tasks are not provisioned on a worst-case basis.

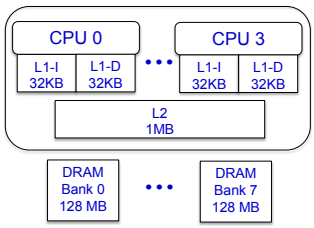


Figure 2: ARM Cortex A9.

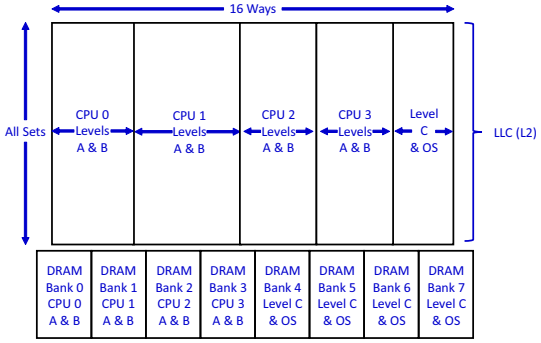


Figure 3: LLC and DRAM allocation.

can also mitigate DRAM interference due to *row-buffer conflicts* [35] by partitioning DRAM banks among task groups.

Fig. 3 depicts the LLC and DRAM allocation strategy used in this paper [9]. 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. Level C is allocated a subsequence of the available LLC ways; this subsequence is used by the OS as well. Level-C tasks are SRT and provisioned on an average-case basis. Under this assumption, Level-C tasks' 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. DRAM is allocated similarly. This partitioning ensures that Level-A and -B tasks do not experience LLC interference from tasks on other cores, *i.e.*, *spatial* isolation. Level-A tasks are also *temporally* isolated from Level-B tasks by being afforded higher priority (this ensures there is no temporally interleaved access to shared cache lines). In considering MC^2-scheduled task systems, we assume all tasks fit in memory and do not incur page faults.

Offline optimization component. The number of LLC ways allocated to the Level-C partition and to the per-core Level-A and -B partitions are tunable parameters. These per-task-set parameters can be determined offline using a linear program that optimizes schedulability [11].

Unmanaged resources. 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 contention sources [46]. However, we assume a measurement-based approach to determining PETs, so such uncontrolled resources are implicitly considered when determining PETs as we measure execution times under the presence of contention for such uncontrolled resources. 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, PETs are often determined via measurement in practice. We assume that sufficient measurements are taken to cover the worst-case behavior of all tasks with respect to unmanaged resources.[3]

3 THE OXYMORON OF "ISOLATED SHARING"

By necessity, any real-time operating system (RTOS) must balance conflicting objectives. Judging from research papers, the primary objective is ostensibly support for predictable timing. The second objective, however, is likely more important: an RTOS *must carry out useful work*. Prior work on MC^2 bridged the gap between these two demands by enabling user-level data sharing between tasks, but this did not go far enough for most real-world use cases. In particular, a system can only be useful if it *produces output*, and this fundamentally requires device I/O. Like IPC, device I/O is a type of data sharing, and, like all forms of data sharing, can cause interference. For example, when Level-A/B tasks use IPC or I/O buffers in Level-C banks, they may suffer cache evictions from Level-C activity. Previously, such interference has been considered in the context of temporal isolation, which is orthogonal to our approach. This section discusses how data sharing, including IPC and I/O, is at odds with the objective of hardware isolation.

3.1 Types of Data Sharing

As mentioned earlier, we extend the data-sharing capabilities of MC^2 in two key areas: *IPC* and *device I/O*.

OS-supported IPC. Previous work on MC^2 specifically focused on data sharing, but took a limited view of the solution: it only supported user-level IPC using shared memory [28]. This limitation facilitates maximizing isolation, because it avoids the need for system calls or dynamic memory allocations. Shared-memory buffers can be allocated and isolated during task initialization, and can be accessed without involving the OS.

However, shared-memory IPC is insufficient in many cases. For example, even simple message-passing systems require in-memory synchronization primitives or wait-free data structures (as the authors of [28] recommend). These shortcomings are addressed by other IPC mechanisms such as message queues or pipes, but these mechanisms break isolation because, in addition to sharing with other tasks, they require sharing data with the OS kernel. Involving the OS kernel is particularly problematic because *the OS is fundamentally shared among all tasks.*

Device I/O. Even if OS-supported IPC can be achieved without compromising isolation, IPC is still only one of many ways in which programs share data. A second major source of data sharing is device I/O. Modern device I/O is largely centered around DMA (direct memory access), where hardware peripherals can directly read from or write to the same DRAM as the CPU. So, to support device I/O, the first requirement is that a real-time system must *prevent unpredictable DRAM interference due to DMA*. However, device interaction does not end with raw data entering DRAM—it must also be processed by user-level CPU tasks. Therefore, a second

[3] Timing analysis for multicore machines is out of scope for this paper. Our measurement based approach is sufficient to inform realistic execution-time behavior under different resource-allocation policies, which are the focus of this work.

requirement is that a real-time system must *deliver I/O data to user-level tasks* (*i.e.*, the OS must place I/O data into userspace memory in order to be accessible to user-level tasks). The second objective is similar to OS-supported IPC because low-level interaction with hardware is typically delegated to the OS kernel, even if I/O is initiated by user-level tasks.

Summary: types of interference due to data sharing. The prior paragraphs established *why* data sharing can lead to hardware interference without describing the specifics of *how* the interference occurs. Fortunately, the types of interference we mitigate in our modifications to MC^2 can be simplified into two categories:

- **CPU-sourced interference.** CPU-sourced interference occurs when tasks suffer interference due to other CPU tasks concurrently accessing a DRAM bank or cache region. Unmanaged IPC and interactions with the OS cause this type of interference.
- **DMA-sourced interference.** DMA-sourced interference occurs when tasks suffer interference due to DMA I/O concurrently accessing DRAM banks. Because DMA bypasses the CPU entirely,[4] DMA-sourced interference causes no cache interference.

Our modifications to MC^2's DRAM and cache-aware memory-management system reduce both of these sources of interference. We note here that other kernel data structures such as task structures, page tables, and page caches are already isolated in Level-C DRAM banks as we described in Sec. 2. We defer a further discussion of our specific modifications until Sec. 4, and instead dedicate the remainder of this section to examples of how these types of interference arise in practice.

3.2 Memory Interference in Real Software

To understand some of the difficulties with data sharing, one can observe the procedures for interacting with devices on a Linux-based system. I/O for different devices can differ surprisingly in terms of software complexity and the potential sources of memory interference. We illustrate this point using two devices: a secondary-storage disk and a USB video camera.

Memory interference from zero-copy I/O. "Zero-copy" refers to I/O that does not require copying data between separate memory buffers. Fig. 4 depicts one possible type of zero-copy I/O in Linux: reading from secondary storage.[5] Generally, a program reads from a disk by issuing a `read` system call and specifying a user-allocated memory buffer to receive the data. This is shown at the top of Fig. 4. This prompts the kernel to determine the specific sector(s) of the disk to read, and to issue a request to the disk via the appropriate communication bus. The data transfer is then actually handled by the disk itself, which will use DMA to populate the user-space buffer. When the DMA transfer is complete, the disk will send an interrupt to the kernel, which returns control to the user task. Finally, the user task is free to operate on the received data.

The zero-copy example from Fig. 4 illustrates two useful points. First, it illustrates both DMA-sourced interference when the device writes to the buffer, and CPU-sourced interference when the task accesses it. Second, being zero-copy, the example involves only a single, user-allocated buffer. Thus, both sources of interference can be managed by properly provisioning a single region of memory.

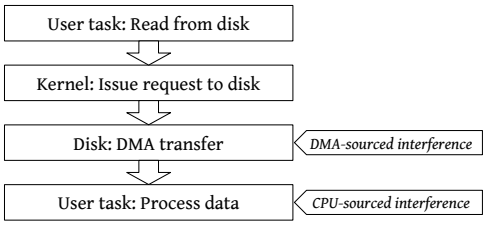


Figure 4: Simplified direct disk I/O data flow.

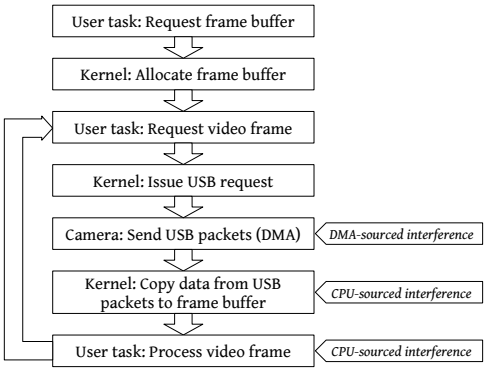


Figure 5: Simplified USB camera I/O data flow.

Memory interference from USB I/O. In contrast to the zero-copy example, Fig. 5 illustrates the flow of data when a user-level task attempts to read a frame from a USB camera on a Linux-based system like MC^2.[6] Tasks using Linux's standard *Video for Linux version 2* (`v4l2`) API must first request one or more buffers to hold video frames, but this actual allocation is managed in kernel code. Next, the user task may issue a request to read a new video frame, which prompts the kernel to start receiving data from the camera. Despite using DMA to transfer USB packets, each USB packet only contains a small portion of the overall frame, which must be copied to the frame buffer. Finally, when the kernel finishes copying an entire frame, the user task is able to access the frame buffer.

Fig. 5 demonstrates the counterintuitive fact that device complexity is not necessarily related to difficulty in preventing interference. In the zero-copy example, ensuring the isolation of a single user-allocated buffer is sufficient to prevent unpredictable CPU- and DMA-sourced interference. However, the seemingly simpler USB webcam driver uses intermediate buffers that can also be subject to CPU- and DMA-sourced interference. An RTOS must instrument allocation decisions by each device driver. Experiments presented in Sec. 5 indicate that the decision as to where to place these dynamic buffers introduces subtle tradeoffs.

Non-data-related sources of interference. While the discussion above focuses on DRAM and cache interference due to *data transfers*, both IPC and I/O also involve other sources of interference, including interrupt overhead and interference due to instruction fetches.

[4] This is the case on our test platform, but may not hold on modern high-end systems.
[5] This actually is not the default disk-access behavior in Linux; zero-copy disk I/O requires passing the optional `O_DIRECT` flag to the `open` system call.
[6] USB devices may not be common in HRT systems; we use a USB camera only as an exemplar of devices where OS activity may cause memory interference.

In practice, however, we found that these sources of interference are small enough to account for by inflating PETs. These overhead-accounting techniques are presented in our prior work [30].

To further reduce the impact of interrupt interference, we redirect all interrupts to a single CPU. To account for interrupt handling time, we inflate the PET of any tasks assigned to this CPU accordingly.[7] As for instruction memory, MC^2 always places kernel code into Level-C banks and cache partitions. This means that Level-A and -B tasks suffer small additional overheads when invoking system calls, which will require instruction fetches from Level-C partitions regardless of where IPC or I/O data buffers are located. This, unfortunately, is an unavoidable consequence of loading kernel code into Level-C banks at boot time—and the MC^2 kernel is too large to fit into a Level-A/B bank.

4 IMPLEMENTATION IN MC^2

The possibility for interference in both I/O and IPC stems from a single difficulty: the need to isolate MC^2's kernel-managed memory buffers. This section discusses our modifications to MC^2's memory-management system, and how we leveraged our modifications to reduce or prevent CPU- and DMA-sourced interference. All of our code, both for these modifications and for running the experiments presented later, is open source.[8]

4.1 Modifications to MC^2 Memory Allocation

As discussed in Sec. 2, MC^2's primary means of reducing shared-hardware interference is cache partitioning and DRAM bank isolation. Level-C tasks may share a dedicated region of the LLC or certain DRAM banks, but Level-A and -B tasks are guaranteed to be free from unpredictable interference whenever possible. Each task is a Linux process, and must invoke a special system call after initialization. This system call prompts the MC^2 kernel to migrate the task's memory to appropriate physical locations [30].

This prior memory-remapping approach has two shortcomings. First, *it only occurs once per task, after initialization.* Second, *it only migrates pages allocated in a task's own address space.* These problems preclude isolating IPC and device I/O, which use dynamically allocated kernel memory. We addressed both shortcomings by modifying the kernel's memory-allocation routines.

Isolating dynamic-memory allocations. Our modifications for isolating dynamic-memory allocations required changes to the *buddy allocator*, a key part of Linux's memory subsystem. The buddy allocator maintains a list of free physical-memory pages. When the kernel allocates memory, either on its own or on behalf of a user-level task, the buddy allocator searches for an appropriately sized free chunk of contiguous physical memory, or a collection of non-contiguous pages if necessary. These pages are then removed from the free list and can be mapped into virtual memory.

Our modification to the buddy allocator consists of replacing the single list of free pages with $m + 1$ independent lists. m of these lists hold free pages for the Level-A and -B tasks on each of the m CPUs. The additional list holds free pages for Level-C tasks. We extended Linux's `get_free_pages` function, which invokes the buddy allocator, to allow specifying which of the $m + 1$ lists to allocate from; by default, we allocate from the Level-C list.

The Linux kernel's `kmem_cache` allocator handles allocations smaller than a single page by subdividing pages obtained from the buddy allocator. Therefore, modifications to the buddy allocator ultimately affect *all* dynamic memory allocations, of both large and small buffers, and in both the kernel and userspace. We also modified the `kmem_cache` allocation routines to use per-core pages for small allocations on behalf of Level-A and -B tasks.

Safe default behavior. A major benefit of the modifications outlined above is that Level-C tasks can dynamically allocate memory without further kernel modification. *Level-C memory allocations, issued by Level-C tasks or by the kernel on behalf of Level-C tasks, no longer cause unpredictable interference in Level-A and -B tasks.* Even though our new allocator is capable of obtaining isolated pages for Level-A and -B tasks, doing so requires explicitly modifying the driver or kernel code where pages are allocated. This is because if high-criticality tasks use IPC or I/O buffers in Level-C banks, they may still suffer cache evictions from Level-C activity. We address such additional complications in Sec. 4.2.

4.2 Optimizing IPC-Related Interference

Even with the kernel modifications described in Sec. 4.1, optimized usage of the newly introduced features requires additional input from MC^2's offline optimization component. Usually, we want to minimize interference experienced by Level-A and -B tasks, which are HRT and are therefore most sensitive to additional overhead. Ideally, Level-A and -B tasks should *only experience interference in the bounded amount of time when they access shared buffers.* The prior work on shared-memory IPC in MC^2 [28] considered this problem in detail—at least for IPC via statically allocated shared buffers. Here, we apply the same proposed techniques to reduce interference in dynamically allocated memory. The techniques are:

- **Selective LLC Bypass (SBP)**: Allocate buffers from Level-C banks, but make them *uncacheable.* Even if tasks concurrently access these buffers, they will not cause other content to be evicted from the LLC, avoiding cache interference.
- **Concurrency Elimination (CE)**: If two communicating tasks are Level-A or -B tasks, assign both to the same CPU, and allocate the buffers from that CPU's Level-A/B bank. Two tasks on the same core cannot concurrently interfere with each other.
- **LLC Locking (CL)**: Lock a pre-allocated buffer into the LLC, so data can be shared without risking evictions or row-buffer conflicts. This approach reduces LLC space for other purposes, but eliminates both cache and DRAM bank interference.

Even though only one of these approaches may be applied to a single given buffer, different approaches can be in use for different buffers across the entire system. To this end, we modified MC^2's offline optimization component to choose the appropriate mechanism for each pair of communicating tasks. We evaluate the efficacy of this optimization in Secs. 5-6.

[7] This requires knowledge of worst-case interrupt interarrival and execution times. We assume that we operate in a "closed world," with *a priori* knowledge of interrupt types and maximum frequencies, as is typically assumed in real-time overhead accounting.
[8] Source code is available at https://wiki.litmus-rt.org/litmus/Publications.

4.3 Optimizing I/O-Related Interference

Because DMA-sourced interference only affects DRAM banks and not the LLC, we explore comparatively fewer management options for I/O buffers. Even so, with the ability to allocate kernel DMA buffers, we can handle DMA-sourced interference in two ways. The first, simpler, way is the default behavior described in Sec. 4.1: place DMA buffers in Level-C banks. The second way is to allocate DMA buffers in a Level-A/B bank if the corresponding device is being used by a Level-A or -B task.

These two approaches have different implications regarding how tasks are impacted by DMA-sourced interference. For example, if DMA buffers are in Level-C banks, then Level-A or -B tasks do not experience row-buffer conflicts as part of DMA-sourced interference. However, as discussed in Sec. 3, I/O data is useless without being accessed, and such accesses can give rise to CPU-sourced interference. Thus, while placing I/O buffers in Level-C banks reduces DMA-sourced interference in Level-A and -B tasks, it will increase CPU-sourced interference when Level-A and -B tasks access those buffers. Similar to the IPC-management options, we incorporated these two I/O-management options into our offline optimization component and evaluate their efficacy in Secs. 5-6.

5 MICRO-BENCHMARK EXPERIMENTS

To assess the impacts of CPU- and DMA-sourced interference under different buffer-allocation options, we experimented with various micro-benchmark programs. We used the results of these experiments to inform the task-system generation process in the schedulability study presented in Sec. 6. In these micro-benchmark experiments, we investigated IPC impacts using Linux's System V message-queue implementation, and I/O impacts using a USB camera and a solid-state disk (SSD). We conducted our experiments assuming an ARM A9 using the allocation scheme in Fig. 3. Our major findings in these experiments are discussed below.

5.1 Impact of IPC-Related Interference

Workloads. To evaluate the SBP, CE, and CL policies from Sec. 4.2, we implemented a *Sender* task, which sends 100 fixed-sized messages, and a *Receiver* task, which receives 100 messages. We ran these tasks at Level A concurrently with a background workload at Level C, with message sizes ranging from 64 to 8,192 bytes. The Receiver was set to start executing after the Sender completes, to guarantee immediate message availability. We designed the background workload to stress the Level-C partition in the LLC and DRAM banks. We measured execution times of `load_msg()`, which allocates a message buffer and copies a message from a user buffer, and `do_msg_fill()`, which copies a message to a user buffer and frees the message buffer. We collected 10,000 samples for each considered message size for each of SBP, CE, and CL. Fig. 6 plots the measured WCET data collected for `load_msg()` and `do_msg_fill()`.

Obs. 1. Sending and receiving execution times were the lowest under CL and the highest under SBP.

Observed CL sending times were between 2.7% and 5.4% of SBP sending times, and receiving times were 10.6% to 15.9% of SBP receiving times. Likewise, CE sending times were between 9.2% and 12.9%, and receiving times were between 29.2% and 40.5%, of the respective SBP times. These results are in accordance with

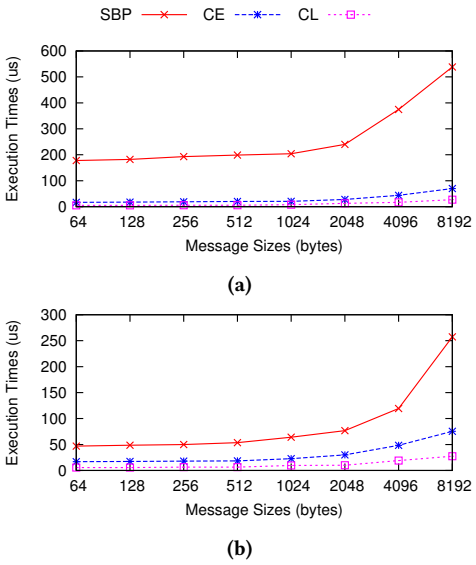


Figure 6: Measured WCETs for `load_msg()`. (a) `load_msg()` and (b) `do_msg_fill()`.

Program	Description
Matrix	DIS Stressmark Suite program. Solve the equation $Ax = b$.
Synthetic	Keep writing arbitrary data to a random memory address.
Framecopy	Copy image data from the frame buffer to user-space buffer.
Yuv2gray	Convert YUV formatted image to a grayscale image.

Table 1: Micro-benchmark programs.

results concerning user-level sharing reported in [10]. CL is the best choice if the LLC is large enough to hold all message buffers. However, CL effectively reduces the LLC size for other purposes. Our new offline component for MC^2 resolves this tradeoff.

5.2 Impact of I/O-Based CPU-Sourced Interference

While IPC only causes CPU-sourced interference, I/O can cause both CPU- and DMA-sourced interference. Of these, we first examine CPU-sourced interference.

Workloads. We assessed CPU-sourced interference using the micro-benchmarks in Tbl. 1. *Matrix* comes from the *data intensive systems (DIS) stressmark suite* [39], which was designed to reflect memory-intensive workloads. *Synthetic* continuously iterates a main loop that writes arbitrary data to randomly selected memory locations. It was designed to stress the LLC, DRAM, and other unmanaged resources. *Framecopy* and *Yuv2gray* are video-processing tasks. We modified the `v4l2` driver, which supports many USB cameras in Linux, to control where buffers are allocated for these tasks.[9] Other non-shared data such as the matrix arrays of Matrix and the array used by Synthetic were statically allocated in accordance with our allocation strategy described in Fig. 3.

Scenarios. To assess CPU-sourced interference, we considered three scenarios:

[9]This was done by adding an extra `GFP(get free pages)` flag to the `kmalloc` interface. We modifed the `v4l2` driver to use this specialized `kmalloc` call.

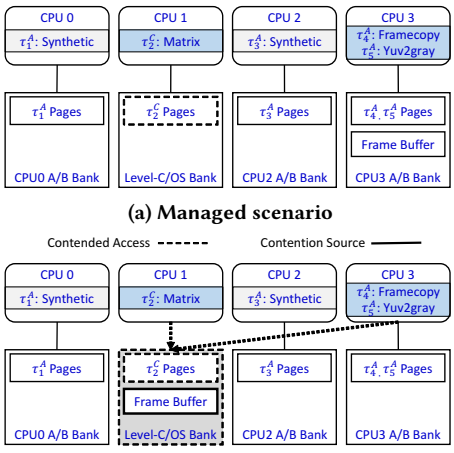


Figure 7: Micro-benchmark tasks and resource allocations.

- Idle: Each micro-benchmark was run alone, with no interfering competing workload.
- Managed: The tasks were executed with the DRAM allocations shown in Fig. 7(a). (Each task's criticality level is denoted with a superscript in this figure.) The frame buffer was allocated in CPU 3's Level-A/B bank. This precludes task τ_4^A and τ_5^A from experiencing CPU-sourced interference.
- Unmanaged: The frame buffer was instead allocated in a Level-C bank, as shown in Fig. 7(b). This causes τ_4^A, τ_5^A, and τ_2^C to experience CPU-sourced interference. This scenario reflects the prior MC^2 implementation, which provides no means for allocating buffers anywhere other than in Level-C banks.

We collected 1,000 execution-time samples for each task in each scenario. Fig. 8 presents normalized (relative to Idle) WCETs obtained from this data for the copy and color-conversion functions of Framecopy (τ_4^A) and Yuv2gray (τ_5^A), respectively, for different frame sizes. We limit measurements to these functions as only these portions of tasks' execution times experience interference due to frame-buffer accesses. Fig. 9 presents data showing the impact on Matrix (τ_2^C) of CPU-sourced interference caused by Yuv2gray (τ_5^A) as a function of the LLC region size allocated to Matrix (τ_2^C). The average-case execution times for Level-C tasks are available online [29].

Obs. 2. CPU-sourced interference inflated the WCET of copying by up to 81%, and of color conversion by up to 30%.

This observation is supported by Fig. 8 and confirms that DRAM interference due to I/O buffers can be problematic. Note that the gap between Idle and Managed is caused by interference from unmanaged resources (see Sec. 2). Under Managed, two instances of Synthetic (τ_1^A and τ_3^A) contend for unmanaged resources, while under Idle, no competing workload exists.

Obs. 3. CPU-sourced interference inflated Matrix's WCET by up to 6%.

This observation is supported by Fig. 9. As expected, the WCET inflation decreases as the number of LLC ways allocated to Matrix increases. The average case (which is of relevance to Level C but

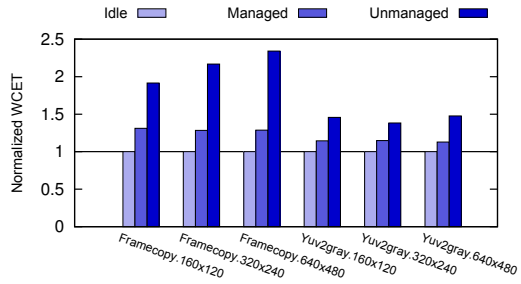


Figure 8: Normalized WCETs of Framecopy and Yuv2gray.

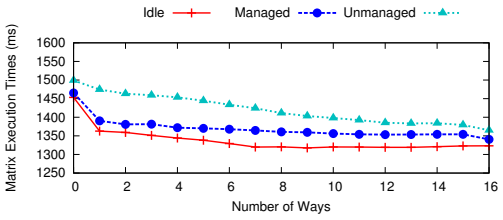


Figure 9: Matrix WCETs as a function of allocated LLC space.

omitted due to lack of space) shows a similar trend. Note that frame-buffer accesses do not break LLC isolation because CPUs 1 and 3 use different partitions in the LLC. The inflation seen here is due to DRAM bank conflicts.

5.3 Impact of DMA-Sourced Interference

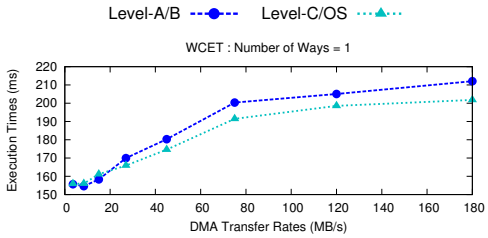
As discussed in Sec. 3, tasks can experience DMA-sourced interference whenever an I/O device sends or reads data. We conducted the following experiments to assess the impact of such interference.

Workloads. We used an SSD to generate DMA transactions at varying bandwidths. This was done by configuring a *Load-Generator* task to repeatedly read 400KB of data from the SSD at a fixed interval. Higher bandwidths were achieved by shrinking the duration of this interval. We configured Load-Generator to access the SSD using the `O_DIRECT` flag to enable DMA directly into user-allocated memory. To observe the impact of DMA-sourced interference, we measured the runtime of a variant of the Synthetic task considered earlier that repeatedly writes 256KB of arbitrary data to randomly selected memory locations.

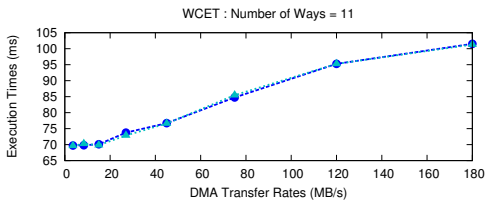
Scenarios. We ran three instances of Load-Generator on separate CPUs, and one instance of Synthetic on the remaining CPU, all at Level A. We collected 1,000 execution-time samples for Synthetic to determine how DMA-sourced interference affected it. This task system was evaluated under two scenarios:

- Level-A/B: All Load-Generator I/O buffers were allocated in Synthetic's Level-A/B bank. This scenario represents the Managed scenario in Sec. 5.2, where an I/O-performing task does not experience CPU-sourced interference, but other tasks on the same CPU could experience DMA-sourced interference.[10]

[10]Note that the Load-Generator tasks were not actually executed on the same CPU as Synthetic. Running them all on the same CPU makes obtaining accurate execution-time measurements more difficult. In the experiments considered here, we are mainly interested in the DMA-sourced interference with which Synthetic must contend, and not the CPUs from which that interference is induced.



(a) Number of LLC ways = 1



(b) Number of LLC ways = 11

Figure 10: Synthetic WCETs under two allocation scenarios.

- Level-C/OS: All Load-Generator I/O buffers were allocated in the Level-C banks. This scenario represents the Unmanaged scenario in Sec. 5.2, where an I/O-performing task experiences CPU-sourced interference.

Fig. 10 presents WCET data obtained for Synthetic under these two scenarios with a small (Fig. 10(a)) and large (Fig. 10(b)) LLC region allocated to it.

Obs. 4. Synthetic's WCET rose with increasing DMA bandwidth.

This observation is supported by both insets of Fig. 10. Because this slowdown happened under both scenarios, we can infer that it is mainly due to memory-bus contention caused by DMA.

Obs. 5. Allocating an I/O buffer in a Level-A/B bank can have a negative impact on other Level-A/B tasks that access that bank.

Fig. 10(a) supports this observation. The difference between the two curves here is due to additional row-buffer conflicts that occur in the Level-A/B scenario. These conflicts become much less of an issue if a task is allocated sufficient LLC space, as seen in Fig. 10(b).

These results expose an interesting tradeoff. If an I/O-performing Level-A or -B task is allocated I/O buffers in its own Level-A/B bank, then it does not experience CPU-sourced interference but the resulting DMA transfers can cause other tasks that access that bank to experience additional row-buffer conflicts as part of DMA-sourced interference. These conflicts can be avoided by allocating I/O buffers in the Level-C banks, but then tasks experience CPU-sourced interference when accessing the buffers. In either case, tasks executing on any CPU can experience memory-bus contention as part of DMA-sourced interference. To sift through this tradeoff and other issues, we conducted a large-scale, overhead-aware schedulability study, which we discuss next.

6 SCHEDULABILITY STUDY

Sec. 5 investigated the impact of our IPC and I/O management approaches on individual tasks, but high-level tradeoffs require investigating potential impacts on entire task systems. To assess such tradeoffs, we conducted a large-scale, overhead-aware schedulability study, which we discuss here.

6.1 Approach

We begin by describing the management schemes that we considered in this study.

Schemes. We considered five management schemes, which vary depending on how buffers for IPC and I/O are handled. To facilitate discussing these schemes, we denote them using the notation $x|y$, where x (resp., y) indicates how IPC (resp., I/O) buffers are handled. The possibilities for x are:

- R (random): IPC buffers are randomly assigned to DRAM banks with none of the optimizations in Sec. 4.2 applied;
- C (C banks used): all IPC buffers are allocated in Level-C banks, again with no optimizations applied;
- O (optimized): IPC buffers are allocated using the optimization techniques in Sec. 4.2.

The possibilities for y are:

- R (random): I/O buffers are randomly assigned to banks;
- C (C banks used): all I/O buffers are allocated in Level-C banks;
- C+A/B (all banks used): an I/O buffer used by a Level-A/B task is assigned to its Level-A/B bank, and those used by Level-C tasks are assigned to Level-C banks.

The five management schemes we considered are: R|R, C|C, O|C, C|C+A/B, and O|C+A/B. We considered R|R to illustrate the ill effects of paying no attention to OS buffer-assignment issues. C|C reflects the choice of partitioning the OS from HRT tasks in a coarse-grained way that prevents OS data structures from existing in a Level-A/B bank (a similar partitioning is possible on most RTOSs today). O|C, C|C+A/B, and O|C+A/B provide varying degrees of fine-grained partitioning in which certain OS data structures are allowed to exist in a Level-A/B bank.

Note that the two choices of C and C+A/B for I/O give two extremes in a spectrum of choices: in the former, all Level-A/B I/O buffers are allocated in Level-C banks, while in the latter, they are all allocated in a Level-A/B bank. It would have been interesting to allow *per-buffer* choices, *i.e.*, allow some Level-A/B I/O buffers to be allocated in Level-C banks and others in a Level-A/B bank. However, while allowing such choices in the context of a single task system is not problematic, doing so is quite impractical at the scale of our schedulability study. Moreover, this change would have greatly complicated explaining our results because it introduces an additional dimension (namely, the fraction of Level-A/B I/O buffers allocated in Level-C vs. Level-A/B banks). Due to space constraints, we have no choice but to avoid such complications. (Given this choice, the "best" schedulability curves we present actually *lower bound* the best that could be obtained from our allocation framework if per-buffer decisions were allowed.)

Modeling IPC and devices. The intent of our study is *not* to delve into complicated precedence-related schedulability issues but rather to demonstrate the effects of DRAM and LLC allocation policies. To avoid such complications, we assumed that tasks that communicate via IPC share a common period, like in prior work on user-level IPC in MC^2 [10]. For similar reasons, we assumed that I/O-consuming Level-A/B tasks simply poll for I/O data (polling is not necessary for Level-C tasks because they are sporadic).

We considered the disk and camera mentioned in Sec. 3.2 as exemplars of two categories of I/O devices. The former only causes

interference when data is pushed by the device or accessed by the task, while the latter involves intermediate steps that cause additional interference. We assumed that intermediate buffers (USB packet buffers in the camera example) remain in Level-C banks under C+A/B to prevent the OS from inducing CPU-sourced interference on Level-A/B tasks while data is copied between buffers.

Task-system generation. We generated task systems by incorporating I/O sources into a procedure used in previous MC^2 work and discussed extensively in prior papers [9–11, 28, 30]. Under this procedure, the following stepwise process is used to generate a task system, and each step is guided by measurement data, *e.g.*, recorded PETs, overheads pertaining to OS activities and I/O, cache reload times, *etc.* (we elaborate on details relevant to this paper below).

(1) Choose distributions from the *first four* categories in Tbl. 2.[11] For example, the *Task Utilization* choice highlighted in bold indicates that tasks generated with that choice will have Level-A tasks with utilizations ranging within $[0.1, 0.2)$. The chosen distributions are used to generate a *preliminary task system*, which is modified below to introduce IPC and I/O.
(2) Select distributions from Category 5 in Tbl. 2. Sample these distributions to determine the size of IPC buffers.
(3) Select a distribution (one for all criticality levels) from Category 6. Sample this distribution to determine the level of I/O bandwidth and assign buffers to tasks until this level is met.
(4) Select distributions from Category 7. Sample these distributions to determine the percentage of I/O tasks whose buffers directly receive data via DMA (like the SSD). The remaining I/O tasks perform intermediate copies (like the USB camera).

Note that the above procedure does not determine task execution costs. A task's execution cost is derived from its period and utilization and represents a bound on the time required for it to complete a job in an idle system with the full LLC available and no other competing work (including I/O). Execution costs under other management options and assumptions are determined using the idle-system cost and micro-benchmark data pertaining to way allocations, I/O buffer allocations, and I/O bandwidths (data like that given in Figs. 9 and 10 is particularly relevant in our context).

Scenarios. We denote each combination of distribution choices using a tuple notation. For example, (C-Light, Moderate, Heavy, . . .) denotes using the C-Light, Moderate, Heavy, *etc.*, distribution choices in Tbl. 2. We call such a combination a *scenario*. We generated sufficient task systems to estimate mean schedulability within ±0.05 with 95% confidence for each scenario and system utilization.

Overhead accounting. Any Level-A/B task that accesses a kernel data structure stored in Level-C banks requires additional execution time. The increases were informed by Fig. 6 for IPC and Fig. 8 for I/O buffers. For the latter, we assumed that each I/O-performing task begins with a system call that copies I/O data from kernel-managed buffers into local buffers. This assumption standardizes the interference to the duration of the copy rather than accounting for all possible buffer-access patterns. Additional CPU-sourced interference also occurs in the R|R scheme, which allows I/O buffers to be allocated in any Level-A/B bank.

We accounted for interrupts for all schemes by inflating PETs of Level-A/B tasks on CPU 0, where interrupts are handled. For Level-C tasks, we used interrupt-accounting techniques from [6], which are applicable under global scheduling.

Category	Choice	Level A	Level B	Level C
1: Criticality Utilization Percent (%)	C-Light	[35, 45)	[35, 45)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	All-Mod.	[28, 39)	[28, 39)	[28, 39)
2: Period (ms)	Short	{12, 24}	{24, 48}	[12, 100)
	Moderate	{20, 40}	{40, 80}	[20, 100)
	Long	{48, 96}	{96, 192}	[50, 500)
3: Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Medium	[0.02, 0.1)	[0.05, 0.2)	[0.1, 0.4)
	Heavy	**[0.1, 0.2)**	[0.2, 0.4)	[0.4, 0.6)
4: Max Reload Time(%)	Quick	[1, 10)	[1, 10)	[1, 10)
	Slow	[25, 50)	[25, 50)	[25, 50)
5: IPC Size (bytes)	Small	{128, 256}	{128, 256}	{128,256}
	Large	{4096, 8192}	{4096, 8192}	{4096, 8192}
6: I/O Bandwidth (MB/s)	Low	[0,20]		
	Medium	[40, 60]		
	High	[180,200]		
7: Direct I/O Tasks(%)	Few	[0, 30]	[0, 30]	[0, 30]
	Many	[70, 100]	[70, 100]	[70,100]

Table 2: Task-set parameters and distributions. $[a, b)$ denotes a continuous interval that is closed on the left and open on the right; $\{E\}$ denotes a discrete set of elements E.

6.2 Results

Our full study generated 648 schedulability plots, one per scenario, taking over 38 days of CPU time to compute. Our full set of plots is available online [29].

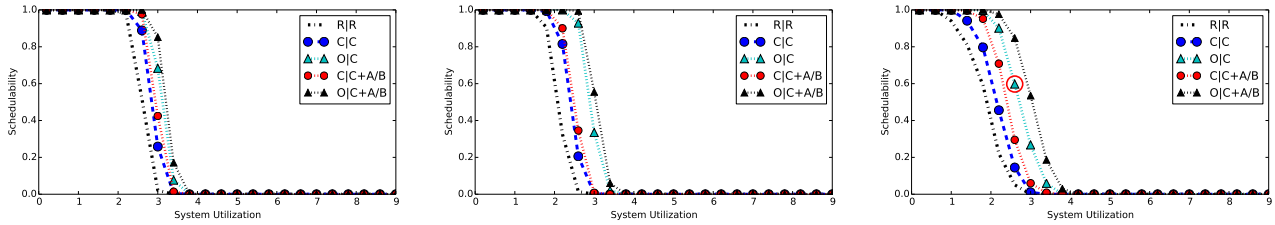
Fig. 11 shows three representative plots. The horizontal axis gives total system utilization.[12] For each utilization, the vertical axis gives the proportion of randomly generated task systems that were schedulable under each considered scheme. For example, the circled point in Fig. 11(c) indicates that 60% of the generated task systems with a total utilization of 2.6 were schedulable under O|C.

Evaluation metric. We used the data in these plots to compute per-scheme *schedulable-utilization areas* (SUAs) in order to compare different schemes. For a given scheme, SUA is simply the sum of the areas under that scheme's curves in all of our 648 plots. A larger SUA implies that a larger fraction of the randomly generated task systems was deemed schedulable, so we can compare any two management schemes by calculating the ratio of their respective SUAs.

Relative impact of various management schemes. We now state several observations that follow from the full set of plots. We illustrate these observations using the plots in Fig. 11.

[11] *Very* briefly (and *informally*), these categories specify: (1) the fraction of the overall workload that exists at each criticality level, (2) task periods, (3) utilizations at each criticality level, and (4) an LLC reload factor used to determine cache-related preemption delays. (2) and (3) are influenced by measurement data so that Level-A, -B, and -C PETs reflect observed inflated worst-case, worst-case, and average-case execution times, respectively. (4) is similarly impacted by measurement data. These details are described in full in previously published papers [10, 11, 28, 30].

[12] The "utilization" referred to here is that initially obtained during task-set generation without accounting for MC^2's hardware management, which improves execution times. Thus, it is possible for a task system to have a total utilization exceeding four and be schedulable.



(a) (C-Light, Long, Light, Quick, Small, High, Many) (b) (C-Light, Long, Light, Quick, Large, High, Many) (c) (C-Light, Short, Med., Slow, Large, High, Many)

Figure 11: Representative schedulability plots.

Obs. 6. Coarse I/O and IPC partitioning is beneficial.

This observation is supported by the fact that the SUA of the C|C scheme is 6% larger than that of R|R. This can be observed qualitatively in Fig. 11, where the R|R curve is always below that for any other management scheme.

Obs. 7. IPC-buffer optimizations outperform coarse partitioning.

The SUA of O|C is 11% greater than that of C|C. This behavior can be observed by comparing the C|C and O|C curves in insets (a) and (b) of Fig. 11. The 11% improvement increases to 13% when only *Large* (Tbl. 2, Category 5) message sizes are considered. As expected, this indicates that the benefit is proportional to the amount of data shared with the OS.

Obs. 8. Reducing CPU-sourced I/O interference is more important than reducing DMA-sourced interference.

This observation addresses the tradeoff discussed in Sec. 4.3. The SUA of C|C+A/B is 3% better than that of C|C. This means that CPU-sourced interference from accessing I/O buffers in Level-C banks is usually worse than the DMA-sourced interference from placing I/O buffers in a Level-A/B bank. The improvement increases to around 4% if we only consider *High* bandwidths (Tbl. 2, Category 6). Unlike in IPC-buffer optimization, reducing CPU-sourced interference in I/O buffers leads to increased DMA-sourced interference, meaning that the overall improvement of I/O-buffer optimization is smaller. However, it is still visible in the C|C+A/B and C|C curves of insets (b) and (c) of Fig. 11.

Obs. 9. Used in conjunction, fine-grained I/O and IPC management outperform all other schemes.

This observation is supported by Fig. 11, where O|C+A/B covers the greatest area in all insets. Quantitatively, the SUA of O|C+A/B is 14% greater than that of C|C, which is even greater than the difference between C|C and R|R. Overall, O|C+A/B exhibits a 21% improvement over the SUA of R|R. From these observations, we can conclude that our extensions to MC^2 significantly improve the schedulability of systems requiring OS-supported IPC and device I/O. In some cases, such as Fig. 11(c), the improvement encompasses nearly an entire core's worth of additional computing capacity. This is even more impressive given that tasks spend only a relatively small proportion of their time accessing IPC and I/O buffers.

7 RELATED WORK

This work's major contribution relates two significant research areas: managing DMA interference, and improving memory allocation to reduce cache and DRAM-bank interference.

Prior research on managing DMA interference includes WCET analysis [20–22], implementing DMA schedulers using hardware mediation [19, 40–42], mixed-criticality systems [37], and scheduling legacy I/O operations [27]. However, these works focus primarily on bus contention, and almost entirely address I/O interference via improved *temporal* isolation. While this is useful, temporal partitioning is orthogonal to our approach, which focuses instead on reducing DRAM bank interference using *spacial* isolation.

Other research has focused on reducing DRAM-bank and cache contention via memory-allocator improvements [33, 51]. Of these, PALLOC [51], like MC^2, uses page coloring to implement a bank-aware memory allocator. (We did not consider page coloring in MC^2 in this work.) However, PALLOC only supports user-level allocations, limiting its applicability to problems related to data sharing with the OS and among user-level tasks.

More broadly, this paper falls within an overarching set of research results pertaining to shared-hardware isolation [31]. Prior efforts have focused on issues such as cache partitioning [3, 5, 18, 26, 43, 49, 50], DRAM controllers [4, 14, 23, 24, 32, 38], and bus-access control [1, 2, 13, 15, 16, 42]. Other work has focused on reducing shared-resource interference when per-core scratchpad memories are used [45], throttling lower-criticality tasks' memory accesses [52], and controlling bandwidth allocations [44].

8 CONCLUSION

We have presented techniques for mitigating OS-induced interference in multicore real-time systems. Our focus on such techniques distinguishes this paper from prior work on providing hardware isolation, which has largely ignored the OS. We evaluated the effectiveness of the considered techniques through micro-benchmark experiments and a large-scale, overhead-aware schedulability study. Our micro-benchmark experiments show that OS-related sharing can increase individual PETs. Our schedulability study demonstrates the importance of properly considering IPC- and I/O-related allocation decisions from a schedulability point of view.

In the future, we plan to apply the work from this paper to systems that must support multiple functional modes (as commonly required in safety-critical systems). In multi-mode systems, the aggregate memory footprint of all tasks may exceed total DRAM capacity. Thus, mode-change protocols may need to dynamically transfer task pages to or from external storage. Such protocols will require OS and I/O support for fast, persistent storage. We are currently investigating the usage of SSDs to meet this need. In other future work, we hope to consider alternative hardware platforms that can help limit DMA interference.

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. Davis. Evaluation of Cache Partitioning for Hard Real-Time Systems. In *ECRTS '14*.

[4] N. Audsley. Memory Architecture for NoC-Based Real-Time Mixed Criticality Systems. In *WMC '13*.

[5] M. A. Awan, K. Bletsas, P.F. Souto, B. Akesson, and E. Tovar. Mixed-Criticality Scheduling with Dynamic Redistribution of Shared Cache. In *ECRTS '17*.

[6] B. Brandenburg, H. Leontyev, and J. Anderson. 2011. An Overview of Interrupt Accounting Techniques for Multiprocessor Real-Time Systems. *J. of Sys. Arch.* 57, 6 (2011), 638–654.

[7] A. Burns and R. Davis. 2014. *Mixed Criticality Systems – A Review.* Technical Report. Department of Computer Science, University of York.

[8] Certification Authorities Software Team (CAST). 2016. Position Paper CAST-32A: Multi-Core Processors. (November 2016).

[9] M. Chisholm, N. Kim, S. Tang, N. Otterness, J. Anderson, F.D. Smith, and D. Porter. Supporting Mode Changes while Providing Hardware Isolation in Mixed-Criticality Multicore Systems. In *RTNS '17*.

[10] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, and F.D. Smith. Reconciling the Tension Between Hardware Isolation and Data Sharing in Mixed-Criticality, Multicore Systems. In *RTSS '16*.

[11] M. Chisholm, B. Ward, N. Kim, and J. Anderson. Cache Sharing and Isolation Tradeoffs in Multicore Mixed-Criticality Systems. In *RTSS '15*.

[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] D. Guo and R. Pellizzoni. A Requests Bundling DRAM Controller for Mixed-Criticality Systems. In *RTAS '17*.

[15] M. Hassan and H. Patel. Criticality- and Requirement-Aware Bus Arbitration for Multi-Core Mixed Criticality Systems. In *RTAS '16*.

[16] M. Hassan, H. Patel, and R. Pellizzoni. A Framework for Scheduling DRAM Memory Accesses for Multi-Core Mixed-Time Critical Systems. In *RTAS '15*.

[17] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS Support for Multicore Mixed-Criticality Systems. In *RTAS '12*.

[18] J. Herter, P. Backes, F. Haupenthal, and J. Reineke. CAMA: A predictable Cache-Aware Memory Allocator. In *ECRTS '11*.

[19] T.-Y. Huang, C.-C. Chou, and P.-Y. Chen. Bounding the Execution Times of DMA I/O Tasks on Hard-Real-Time Embedded Systems. In *RTCSA '03*.

[20] T.-Y. Huang, C.-C. Chou, and P.-Y. Chen. 2006. Bounding DMA Interference on Hard-Real-Time Embedded Systems. *J. Inf. Sci. Eng.* 22 (09 2006), 1229–1247.

[21] T.-Y. Huang, J. W.-S. Liu, and J.-Y. Chung. Allowing cycle-Stealing Direct Memory Access I/O Concurrent with Hard-Real-Time Programs. In *ICPADS '96*.

[22] T.-Y. Huang, J. W.-S. Liu, and D. Hull. A Method for Bounding the Effect of DMA I/O Interference on Program Execution Time. In *RTSS '96*.

[23] 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*.

[24] 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*.

[25] 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*.

[26] 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*.

[27] J. Kim, M. Yoon, R. Bradford, and L. Sha. Integrated Modular Avionics (IMA) Partition Scheduling with Conflict-Free I/O for Multicore Avionics Systems. In *COMPSAC '14*.

[28] N. Kim, M. Chisholm, N. Otterness, J. Anderson, and F.D. Smith. Allowing Shared Libraries while Supporting Hardware Isolation in Multicore Real-Time Systems. In *RTAS '17*.

[29] N. Kim, S. Tang, N. Otterness, J. Anderson, F.D. Smith, and D. Porter. 2018. Supporting I/O and IPC via Fine-Grained OS Isolation for Mixed-Criticality Real-Time Tasks. Full version of this paper, available at http://jamesanderson.web.unc.edu/papers/. (2018).

[30] N. Kim, B. Ward, M. Chisholm, J. Anderson, and F.D. Smith. 2017. Attacking the One-Out-Of-m Multicore Problem by Combining Hardware Management with Mixed-Criticality Provisioning. *Real-Time Sys.* 53, 5 (2017), 709–759.

[31] 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*.

[32] Y. Krishnapillai, Z. Wu, and R. Pellizzoni. ROC: A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems. In *ECRTS '14*.

[33] X. Liao, R. Guo, H. Jin, J. Yue, and G. Tan. 2017. Enhancing the Malloc System with Pollution Awareness for Better Cache Performance. *IEEE Trans. Parallel Distrib. Syst.* 28, 3 (March 2017), 731–745.

[34] LITMUS^{RT} Project. 2018. (2018). http://www.litmus-rt.org/.

[35] 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*.

[36] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed Criticality Real-Time Scheduling for Multicore Systems. In *ICESS '10*.

[37] D. Münch, M. Paulitsch, and A. Herkersdorf. Temporal Separation for Hardware-Based I/O Virtualization for Mixed-Criticality Embedded Real-Time Systems Using PCIe SR-IOV. In *ARCS '14*.

[38] S. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. 2011. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *MICRO '11*.

[39] J. Musmanno. 2003. Data Intensive Systems (DIS) Benchmark Performance Summary. (Aug. 2003).

[40] R. Pellizzoni, B. Bui, and M. Caccamo. Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems. In *RTSS '08*.

[41] R. Pellizzoni and M. Caccamo. 2010. Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems. *IEEE Trans. Comput.* 59 (March 2010), 400–415.

[42] R. Pellizzoni, A. Schranzhofer, J. Chen, M. Caccamo, and L. Thiele. Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *DATE '10*.

[43] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. 2016. A Software Cache Partitioning System for Hash-Based Caches. *ACM Trans. Archit. Code Optim.* 13, 4, Article 57 (Dec. 2016), 24 pages.

[44] G. Seetanadi, J. Cámara, L. Almeida, K. Årzén, and M. Maggio. Event-Driven Bandwidth Allocation with Formal Guarantees for Camera Networks. In *RTSS '17*.

[45] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. Phatak, R. Pellizzoni, and M. Caccamo. A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems. In *RTAS '16*.

[46] P. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *RTAS '16*.

[47] S. Vestal. Preemptive Scheduling of Multi-Criticality Systems with Varying Degrees of Execution Time Assurance. In *RTSS '07*.

[48] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making Shared Caches More Predictable on Multicore Platforms. In *ECRTS '13*.

[49] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee. Analysis and Implementation of Global Preemptive Fixed-Priority Scheduling with Dynamic Cache Allocation. In *RTAS '16*.

[50] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee. vCAT: Dynamic Cache Management using CAT Virtualization. In *RTAS '17*.

[51] H. Yun, R. Mancuso, Z. Wu, and R. Pellizzoni. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicoore Platforms. In *RTAS '14*.

[52] 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*.