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 Donald E. Porter

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 also 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.

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 (Erickson et al. 2015; Kim et al. 2017b): 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 (Certification Authorities Software Team 2016), 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 (Alhammad and Pellizzoni 2016; Alhammad et al. 2015; Altmeyer et al. 2014; Audsley 2013; Chisholm et al. 2016, 2015; Giannopoulou et al. 2013; Hassan and Patel 2016; Hassan et al. 2015; Herter et al. 2011; Jalle et al. 2014; Kim et al. 2015, 2014a, 2013, 2017a,b; Kotaba et al. 2013; Krishnapillai et al. 2014; Pellizzoni et al. 2010; Tabish et al. 2016a; Ward et al. 2013; Xu et al. 2016; Yun et al. 2014, 2012).

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.

An ideal, "complete" solution to the one-out-of-m problem would require preventing interference from all possible sources of sharing in a system. In reality, this ideal solution is impossible—unfortunately, eliminating all interference

*Work supported by NSF grants CNS 1409175, CPS 1446631, CNS 1563845, CNS 1717589, ARO grant W911NF-17-1-0294, ONR grant N00014-20-1-2698, and funding from General Motors.

is only tenable if 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 the negative side effects. Prior work in this direction by our group has addressed user-level data sharing (Chisholm et al. 2016) and the usage of shared libraries (Chisholm et al. 2016; Kim et al. 2017a) in the context of a hardware-isolation framework targeting mixed-criticality systems called MC^2 (mixed-criticality on multicore) (Chisholm et al. 2017, 2016, 2015; Herman et al. 2012; Kim et al. 2017a,b; Mollison et al. 2010; Ward et al. 2013). To our knowledge, however, no prior work on the one-out-of-m problem has investigated techniques specifically for lessening the impact of OS-induced sharing by optimizing memory allocations.

Contributions. The primary contributions of this paper are:

- to provide a comprehensive description and explanation of the tensions between task isolation and data-sharing among tasks as mediated by the OS through I/O and IPC;
- to propose a set of memory-management strategies that favorably resolve this tension;
- and most importantly, to provide an extensive evaluation of the design space and trade-offs pertaining to these memory-management strategies based on enhancing real-time schedulability.

Our investigation required advances 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 is still available 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. In these experiments, we 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. The rest of this paper is organized as follows. We begin in Sec. 2 by providing needed background. Then, in Sec. 3, we describe how OS-induced sharing introduces interference, and in Sec. 4, we describe how we can mitigate such interference. Next, in Sec. 5, we describe our micro-benchmark experiments, and in Sec. 6, we discuss our schedulability study. Finally, we discuss related work in Sec. 7 and conclude in Sec. 8.

2 Background

In this section, we present relevant background material.

Task model. 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.

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

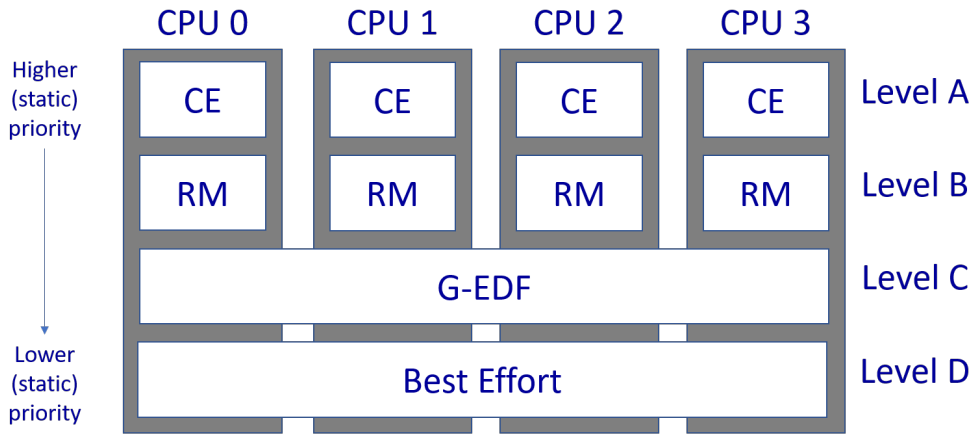


Figure 1 Scheduling in MC^2 on a quad-core machine.

Mixed-criticality scheduling. For systems with tasks of differing criticalities, Vestal proposed *mixed-criticality (MC)* schedulability analysis, which uses less-pessimistic execution-time provisioning for less-critical tasks (Vestal 2007). 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 (2007), 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 on MC scheduling and analysis, surveyed in Burns and Davis (2019). Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors (Mollison et al. 2010).

MC^2 is implemented under LITMUS$^{\mathrm{RT}}$ (LITMUS$^{\mathrm{RT}}$ Project 2018), 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. In this paper, we require that Level-A and -B tasks use polling when waiting for I/O or IPC, because dealing with suspensions greatly complicates both scheduling and schedulability analysis, and delving into the nuances of such analysis is beyond the scope of this work.

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 (Kim et al. 2017b), we assume that Level-B and -C PETs are determined to be, respectively, maximum observed execution times (MOETs) and average-case execution times, and that Level-A PETs are obtained by inflating Level-B PETs by 50%.

MC^2 hardware platform. MC^2 includes several mechanisms for managing the LLC and DRAM banks (Kim et al. 2017b), several of which require additional hardware support provided by MC^2's test platform: the NXP i.MX6 quad-core ARM Cortex A9 evaluation board. The rest of the paper assumes the same test platform. Each core on this machine is clocked at 800MHz, supports out-of-order execution, 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.

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

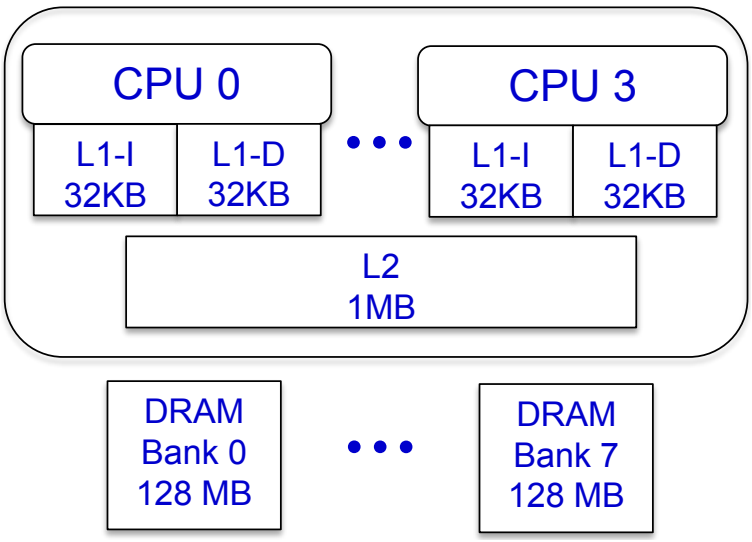


Figure 2 ARM Cortex A9.

Bus controller configuration of the i.MX6. The bus system on the NXP i.MX6 consists of five NIC-301 AXI (Advanced eXtensible Interface) arbiters (ARM Limited 2010), and each arbiter contains one or more bus switches. The arbiter uses an arbitration policy that allocates a single bus cycle at a time to a contending device based on the Quality-of-Service (QoS) settings for the devices, discussed in more detail below. Under default QoS settings, the arbiter allocates each bus cycle to contending devices using a Least-Recently-Used (LRU) algorithm. The system has a single SATA II interface, which is connected to one of the five arbiters. We configured the system memory controller to disable bank interleaving (Freescale 2014, page 3876), as this was necessary to support bank partitioning, but did not modify any other memory-controller settings.

On the surface, the availability of bus-level QoS features may sound like good news for a project seeking to reduce memory-bus interference. Upon close examination, however, we unfortunately found that the bus arbiters' QoS settings were unable to meet our needs for two reasons. First, the arbiter's QoS setting is only one of several factors affecting memory-access priority within the memory controller (Freescale 2014, page 3813). Second, the QoS settings apply to all cores simultaneously,[3] complicating any attempt to prioritize memory access on a per-task or per-core basis using the hardware-supported QoS settings. Therefore, in this paper, we opted not to use the bus arbiter's prioritization settings to further manage DMA (direct memory access) I/O, and left all of the bus arbiter's settings in their default state.

Cache and DRAM management in MC^2. Our platform is an I/O coherent system that has an Accelerator Coherency Port (ACP). Accesses to shared DMA memory regions are routed to the cache controller, which does not insert new data into the cache, but instead only invalidates or cleans cache lines in order to ensure coherency (ARM Limited 2009).

MC^2 supports LLC management using per-core *lockdown registers* to assign ways (columns) of the LLC to task groups (Kim et al. 2017b). It can also allocate sets (rows) of the LLC to task groups for finer partitioning, but we do not explore this option here. Of more importance to this paper is MC^2's ability to partition DRAM banks among task groups in order to mitigate DRAM interference due to *row-buffer conflicts* (Liu et al. 2012).

Fig. 3 depicts the LLC and DRAM allocation strategy used in this paper. 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-core 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

[3] All cores share two separate connections to the bus arbiter (Freescale 2014, page 3982), but assigning a separate QoS value to each connection would be largely meaningless because, to our knowledge, there is no way to specify which of the two connections any given memory request will use.

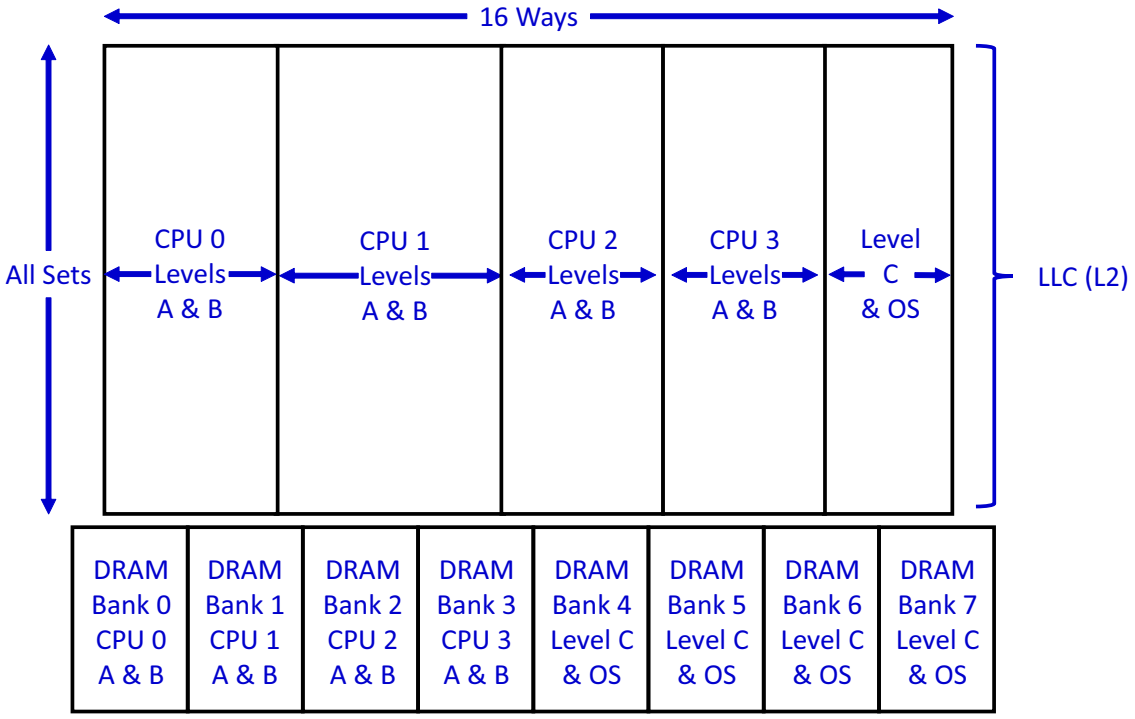


Figure 3 LLC and DRAM allocation. The boundaries within the LLC are configurable parameters.

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.

Unmanaged hardware resources. The MC^2 implementation described in the preceding paragraphs does not provide management for L1 caches, translation lookaside buffers (TLBs), memory controllers, memory buses, or cache-related registers that can be contention sources (Valsan et al. 2016). 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.[4]

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 (Chisholm et al. 2015).

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*. Early work on MC^2 focused on the first objective, and prevented cache and DRAM bank interference between concurrent tasks. Subsequent MC^2 variants began to bridge the gap between the two demands of timeliness and usefulness by enabling user-level data sharing between tasks. Data sharing allowed tasks to interact with each other, but, unfortunately, inter-task data sharing alone is not 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

[4]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.

IPC, device I/O is a type of data sharing, and it, 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 DRAM banks, they may suffer cache evictions from Level-C activity. In prior work, such interference has been considered in the context of temporal isolation (Pellizzoni et al. 2008b; Pellizzoni and Caccamo 2010; Kim et al. 2014b; Muench et al. 2014), 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, here 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 (Kim et al. 2017a). 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 often less convenient or less efficient than OS-supported IPC. For example, even simple message-passing systems require in-memory synchronization primitives or wait-free data structures (as recommended in Kim et al. (2017a)). 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 several 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, where hardware peripherals can directly read from or write to the same DRAM as the CPUs. Therefore, to support device I/O, the first requirement is that a real-time system must prevent or account for unpredictable DRAM interference due to DMA. However, device interaction does not end with raw data entering DRAM—such data must also be processed by user-level CPU tasks. Therefore, a second 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 user-space 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 (we do not consider cases where I/O is entirely handled in user space).

Summary: types of interference due to data sharing. The prior paragraphs establish *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 the same DRAM bank or cache region. Kernel-allocated IPC buffers that do not conform to our LLC and DRAM allocation policy and interactions with the OS cause this type of interference.
- **DMA-sourced interference.** DMA-sourced interference occurs when tasks suffer interference due to ongoing DMA transfers. In this paper, we specifically address DMA-sourced DRAM bank interference. In contrast to CPU-sourced interference, DMA-sourced interference has only a minor impact on our Cortex-A9 platform's LLC.[5] DMA may interfere with other resources not managed in this work, including TLBs, memory buses, memory controllers, and cache-related registers.

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.

[5]By this, we mean that interference due to cache *evictions* does not occur on our platform as the DMA data pages are marked as uncacheable. However, as mentioned in Sec. 2, overhead due to the coherency protocol (*i.e.* invalidating cache lines) may still exist.

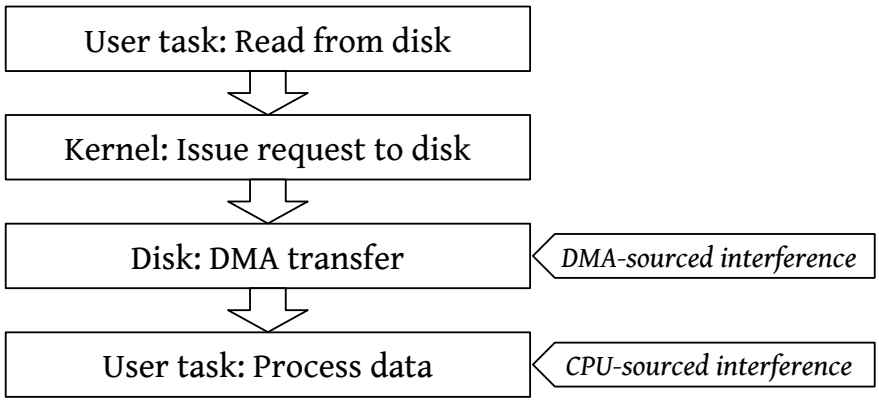


Figure 4 Simplified direct disk I/O data flow.

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.[6] 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 sectors of the disk to read, and to issue a request to the disk via the appropriate communication bus. The disk itself will then 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, this example involves only a single, user-allocated buffer. Therefore, both sources of interference can be managed by properly provisioning a single region of memory. For example, if we allocate the buffer in a Level-A/B DRAM bank, then a Level-A/B task performing I/O does not experience CPU-sourced interference, but all Level-A/B tasks allocated on this bank may suffer DMA-sourced interference due to bank conflicts. On the other hand, if we allocate the buffer in the Level-C banks, then the I/O-performing task experiences CPU-sourced interference, but Level-A/B tasks will not experience DMA-sourced interference.

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.[7] 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 driver uses intermediate buffers that can also be subject to CPU- and DMA-sourced interference. An RTOS must instrument

[6] 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.

[7] 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.

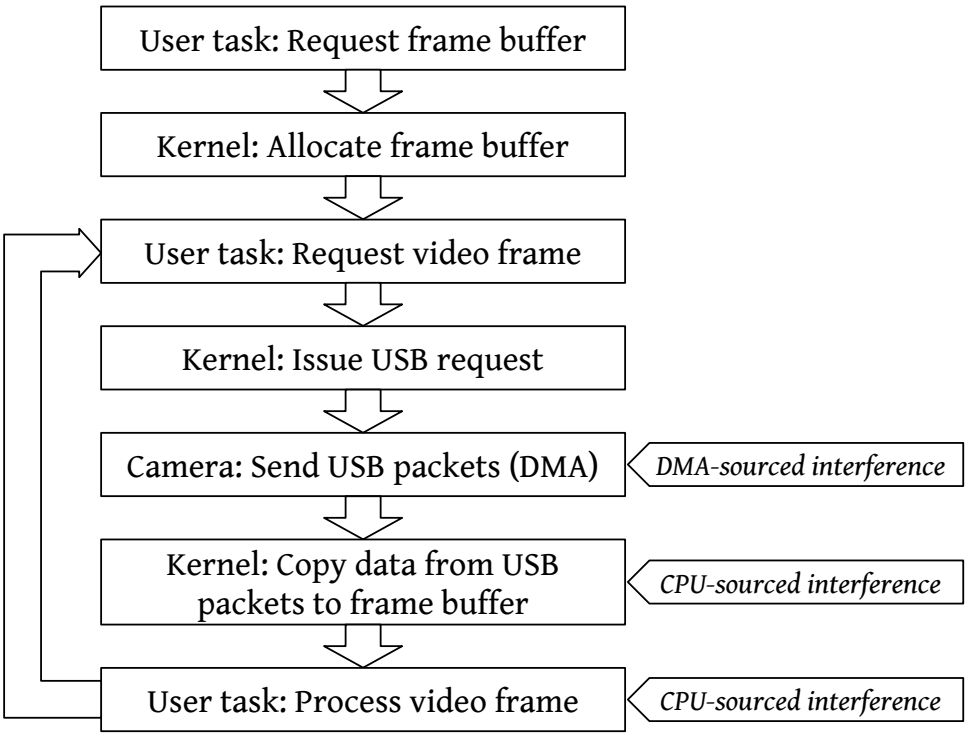


Figure 5 Simplified USB camera I/O data flow.

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 device I/O also involve other sources of interference, including interrupt overhead and interference due to instruction fetches. In practice, however, we found that these sources of interference are small enough to account for by inflating PETs. We leverage overhead-accounting techniques presented in our prior work (Kim et al. 2017b) to accommodate this requirement.

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.[8] As MC^2 always places kernel code into the Level-C banks and LLC partition, the OS accesses Level-C DRAM and LLC areas when it executes interrupt handlers or invokes system calls. Therefore, Level-A and -B tasks assigned to the interrupt-handling processor do not experience interference with respect to the LLC and memory banks. However, when Level-A and -B tasks on this processor invoke system calls, they suffer a small amount of additional interference because the OS fetches instructions from the Level-C banks and LLC 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.[9]

[8] 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.

[9] Source code is available at `https://wiki.litmus-rt.org/litmus/Publications`.

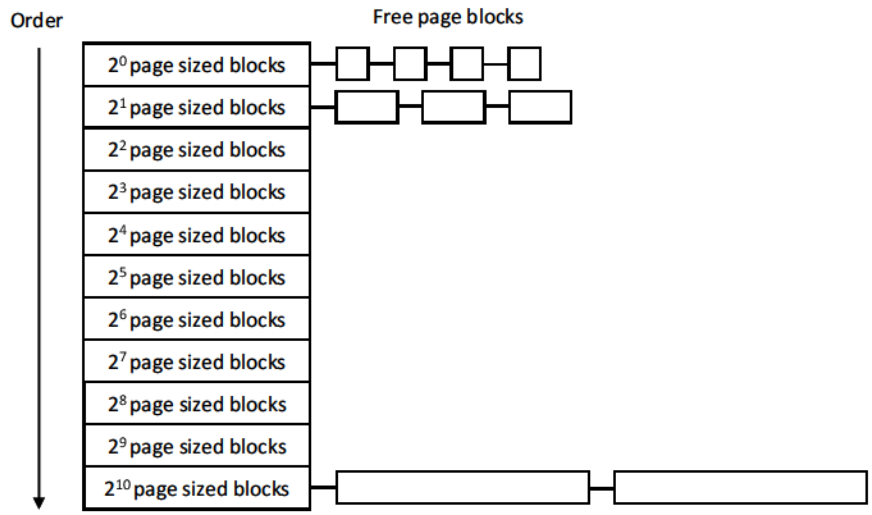


Figure 6 A list of free blocks managed by the buddy allocator.

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 (Kim et al. 2017b).

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. We begin by describing the memory-management system in Linux. We then present our modifications to this system.

Memory management in Linux. In Linux, physical pages are managed and allocated by the *buddy allocator* (Knowlton 1965; Knuth 1968). Memory is divided into blocks of pages where the size of each block is 2^k pages for some k. The buddy allocator maintains a list of free blocks.[10] Fig. 6 illustrates the management of free blocks in Linux. When the OS kernel allocates memory, the buddy allocator searches for an appropriately sized block. When a block of the requested size is not available, a larger block is divided into two half-sized blocks. (These two blocks are "buddies" to each other—hence the name "buddy allocator.") One of these halves is used for the allocation and the other is simply added back into the list of free blocks. When a block is freed, it will be re-combined with its buddy if the buddy is still free.

Our modification to the memory-management system. Our modification to the buddy allocator consists of replacing the single list of free pages with m+1 independent lists as illustrated in Fig. 7. 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 core allocation function, `__alloc_pages()`, which is used internally by all of Linux's memory-allocation APIs, to allow specifying which of the m+1 lists to allocate from. By default, we allocate from the Level-C list.

With this modification, all memory requests are served by the Level-C/OS DRAM banks unless an allocation request explicitly requires Level-A/B pages. To enable such requests, we extended the *Get Free Page (GFP)* flags, a set of bits that must be provided when invoking most memory-allocation routines in the Linux kernel. Normally, these flags configure several aspects of how Linux's memory allocator behaves for an allocation. Our modification only required

[10]The buddy allocator maintains a list of free blocks per *zone*. A zone is a group of pages that have similar properties. The hardware platform considered in this paper has only one zone, `ZONE_NORMAL`. However, other architectures may have multiple zones such as `ZONE_DMA`, `ZONE_NORMAL`, and `ZONE_HIGHMEM`. In such platforms, the buddy allocator can have more than one list (Gorman 2004).

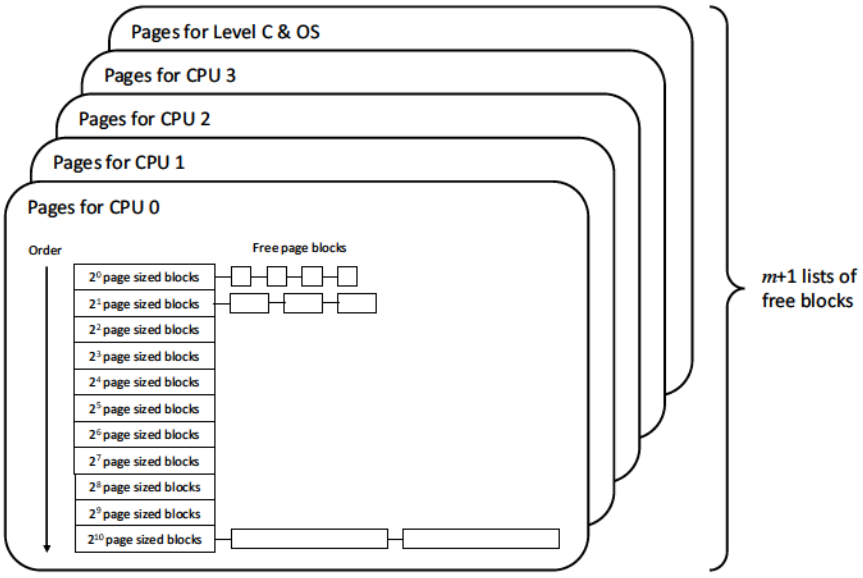


Figure 7 Modified m+1 lists of free blocks in the buddy allocator.

adding a single bit to the GFP flags. If this bit is set, it prompts our modified buddy allocator to determine which processor is making the request, and to fulfill the request from the free-page list for that processor's Level-A/B bank.

As each CPU core has one dedicated 128 MB DRAM bank for high-criticality Level-A and -B tasks, we are unable to allocate more than 128 MB to any group of Level-A and -B tasks that share a CPU core (as doing so would require breaking isolation by allocating from other cores' banks). We assume the total memory consumption of Level-A and -B tasks fits within this constraint and, in our implementation, any attempt to get free pages from such a list when none are available will fail.

Device drivers and IPC system calls generally request kernel memory via two simple interfaces, `vmalloc` and `kmalloc`. The `vmalloc` function calls the `__alloc _pages` function, so we can achieve criticality-aware `vmalloc` allocations by extending the `__alloc_pages` function as mentioned earlier. However, `vmalloc` is typically used to allocate larger chunks of memory. The `kmalloc` function, by contrast, is used when smaller memory buffers are required. As the kernel manages the system's physical memory in page-sized chunks, the `kmalloc` function may use a different memory-allocation technique to provide sub-page-sized allocations. To handle small memory allocations, Linux uses a set of pools of memory objects of fixed sizes; this functionality is provided by the `kmem_cache` allocator. Therefore, we also extended this `kmem_cache` allocator to create per-core memory pools with Level-A/B pages in order to serve small memory allocations on behalf of Level-A and -B tasks. The `kmem_cache` allocator creates memory pools of different sizes for `kmalloc` allocations at boot time 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 user space.

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. Unfortunately, simply allowing Level-A and -B tasks completely unfettered access to IPC or I/O buffers in Level-C banks may cause them to suffer cache evictions from Level-C activity. We address such additional complications in Sec. 4.2.

4.2 Optimizing Allocations to Reduce IPC-Related Interference

Even with the kernel modifications described in Sec. 4.1, making the best use 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 a bounded amount of time when they access shared buffers*. The prior work on shared-memory IPC in MC^2 (Kim et al. 2017a) considered this problem in detail—at least for IPC via statically allocated shared buffers.[11] 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. This prevents DRAM bank and LLC interference from tasks running on other CPUs. Cache-reload overheads are already accounted for in our analysis for Level-B tasks, and Level-A tasks cannot be preempted by Level-B tasks so they do not experience cache reload overheads.
- **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.

4.3 Optimizing I/O-Related Interference

DMA-sourced interference primarily affects DRAM banks and not the LLC, so 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. If all 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, whereas allocating buffers from per-core Level-A/B banks may expose the Level-A or -B tasks on that core to 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. So, 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.

4.4 Summary of Memory-Allocation Options

Table 1 summarizes the potential buffer-allocation options enabled by our modifications to MC^2. The columns of Table 1 are divided based on the location of the buffer (which may be in the Level-A/B bank for a given core, or the Level-C banks shared among all cores), and the criticality level of the task accessing the buffer. The rows of Table 1 indicate both whether such an allocation causes the task for which the buffer is allocated to experience interference, and whether such an allocation will cause interference in other tasks.

Table 1 summarizes several facts described earlier in the paper. First, no task can cause or experience CPU-sourced interference from other tasks assigned to the same core (preemptions do not fall into the definition of CPU-sourced interference from Sec. 3.1; we account for them elsewhere in analysis), so the third and fourth rows of the table are all "No". The rightmost column, representing Level-C tasks, contains "Yes" in every row apart from the third and fourth, because buffers for Level-C tasks can only be allocated from Level-C banks, which are always subject to

[11] The prior work also considers sharing between tasks of different criticality levels, but we chose not to evaluate such sharing in this paper to reduce the complexity of the schedulability study, which already required the addition of several new parameters. However, cross-criticality sharing remains theoretically possible under our new modifications so long as it remains limited to wait-free communication.

	Task's Criticality Level	A		B		C
	Task's Buffer Location	A/B	C	A/B	C	C
On/from the same CPU	Causes DMA-sourced	Yes	Yes†	Yes	Yes	Yes
	Experiences DMA-sourced	Yes	Yes*	Yes	Yes*	Yes
	Causes CPU-sourced	No	No	No	No	No
	Experiences CPU-sourced	No	No	No	No	No
On/from other CPUs	Causes DMA-sourced	No	Yes†	No	Yes	Yes
	Experiences DMA-sourced	No	Yes*	No	Yes*	Yes
	Causes CPU-sourced	No	Yes	No	Yes	Yes
	Experiences CPU-sourced	No	Yes*	No	Yes*	Yes

* Only experiences interference when accessing the particular buffer

† Only causes interference to Level-C tasks

Table 1 Types of interference a task may cause or experience, depending on whether it has a buffer located in a Level-A/B bank or Level-C banks.

interference from other Level-C tasks on all cores. In contrast, MC^2's bank isolation ensures that Level-A/B tasks will never experience interference from other cores if all of their buffers are located in the per-core Level-A/B bank.

Entries marked "Yes*" correspond to the cases where Level-A/B tasks allocate I/O or IPC buffers in Level-C banks. As discussed earlier, even though these cases may experience additional interference, the policies discussed in Secs. 4.2 and 4.3 ensure that interference is only experienced when accessing the buffer, differentiating these cases from the other "Yes" entries in the table, where interference can occur at any time during the task's execution. Likewise, entries marked "Yes†" correspond to cases where Level-A/B tasks allocate I/O buffers in Level-C banks. These entries are distinct from others marked "Yes" in that such allocations only cause Level-C tasks to experience additional interference. On the other hand, other Level-A/B tasks, which require stricter isolation, are not affected by bank conflicts due to these allocations.

While Table 1 summarizes the causes and effects of various types of interference, it does not reflect the relative magnitude of each type of interference. We investigate performance impacts in the following section, using micro-benchmark experiments.

5 Micro-Benchmark Experiments

To assess the impacts of CPU- and DMA-sourced interference under the different buffer-allocation options, we experimented with various micro-benchmark programs. The micro-benchmark experiments here serve a dual purpose. First, they simply provide a concrete illustration of interference's negative impact on timing. Second, they provide the inflation factors used during the task-system generation process, required by the schedulability study 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 on the ARM A9 platform described in Sec. 2 using the allocation scheme shown in Fig. 3.

In cases where we computed average-case execution times (ACETs), we used the arithmetic mean to compute averages. Across all of our means, the largest confidence interval (at a 95% confidence level) was 11 microseconds—a negligible time in those experiments, which had measured response times of hundreds of milliseconds. We continue below with our major experimental findings.

5.1 Impact of IPC-Related Interference

Workloads. To evaluate the SBP, CE, and CL policies from Sec. 4.2 on IPC, we implemented tasks that communicated using message queues. Specifically, 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

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 2 Micro-benchmark programs.

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 the kernel functions `load_msg()` and `do_msg_fill()`. The `load_msg()` function allocates a message buffer and copies a message from a user buffer, and `do_msg_fill()` copies a message to a user buffer and frees the message buffer. We collected 10,000 samples for each considered message size under each of the three management strategies (SBP, CE, and CL). Fig. 8 plots the MOET data collected for `load_msg()` and `do_msg_fill()`.

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

Unsurprisingly, the time required to send and receive messages was the fastest when IPC buffers were locked into the cache (CL) and slowest when they were forced to bypass the cache (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. The 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 results concerning user-level sharing reported in Chisholm et al. (2016). CL offers the fastest response times if the LLC is large enough to hold all message buffers. However, CL effectively reduces the LLC size for other purposes. CE provides a good middle ground, as two Level-A or -B tasks assigned to the same core can communicate without any interference from the Level-C background task. CE exhibits only slightly slower execution times than CL, likely due to the fact that its communication buffers are no longer locked into the cache, meaning that the buffers can sometimes be evicted by the communicating tasks' other activities. Our new offline component for MC^2, detailed in Sec. 6, takes these tradeoffs into consideration.

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

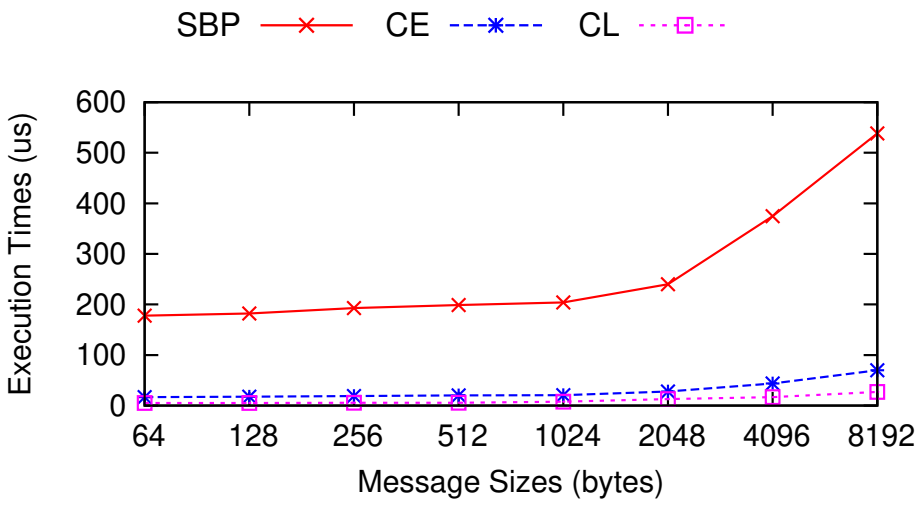
While IPC only causes CPU-sourced interference, I/O can cause both CPU- and DMA-sourced interference. Of these, we first examine CPU-sourced interference.

Workloads. We assessed CPU-sourced interference using the micro-benchmarks in Table 2. *Matrix* comes from the *data intensive systems (DIS) stressmark suite* (Musmanno 2003), 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.[12] 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.

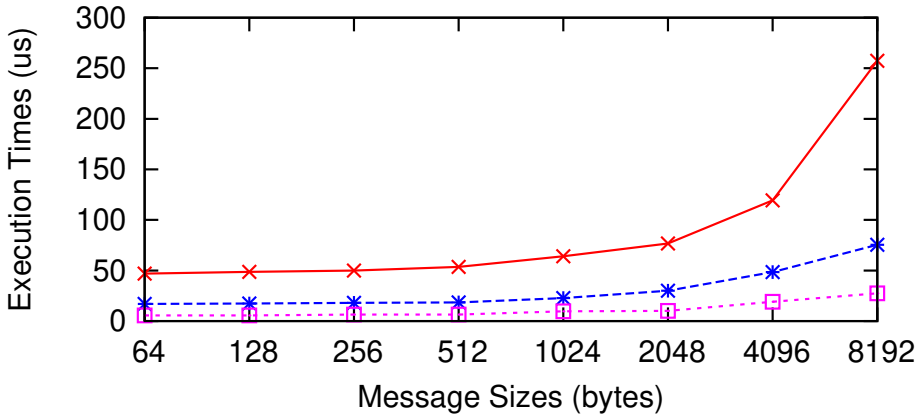
Scenarios. To assess CPU-sourced interference, we considered three scenarios:

- Idle: Each micro-benchmark was run alone, with no interfering competing workload.
- Managed: The tasks were executed with the DRAM allocations shown in Fig. 9(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 prevents tasks τ_4^A and τ_5^A from experiencing CPU-sourced interference.

[12] This was done by modifying calls to `kmalloc` and similar functions to allocate pages from the per-core high-criticality partitions when necessary via the memory-management interface changes described in Sec. 4.

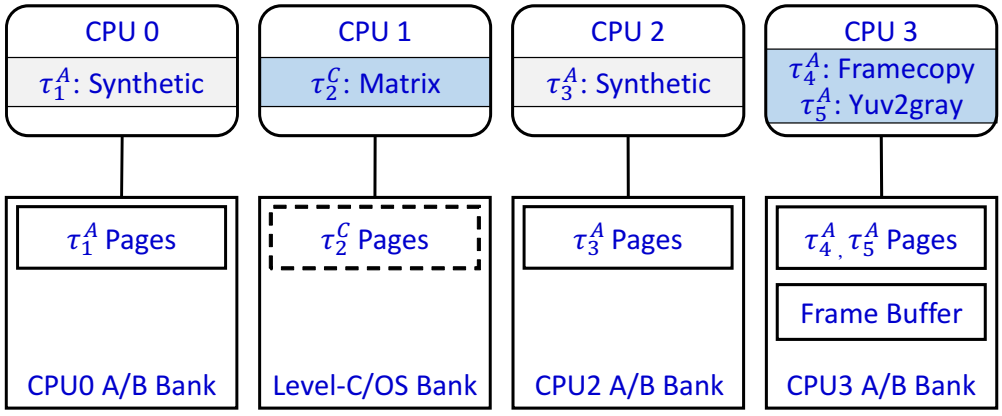


(a)

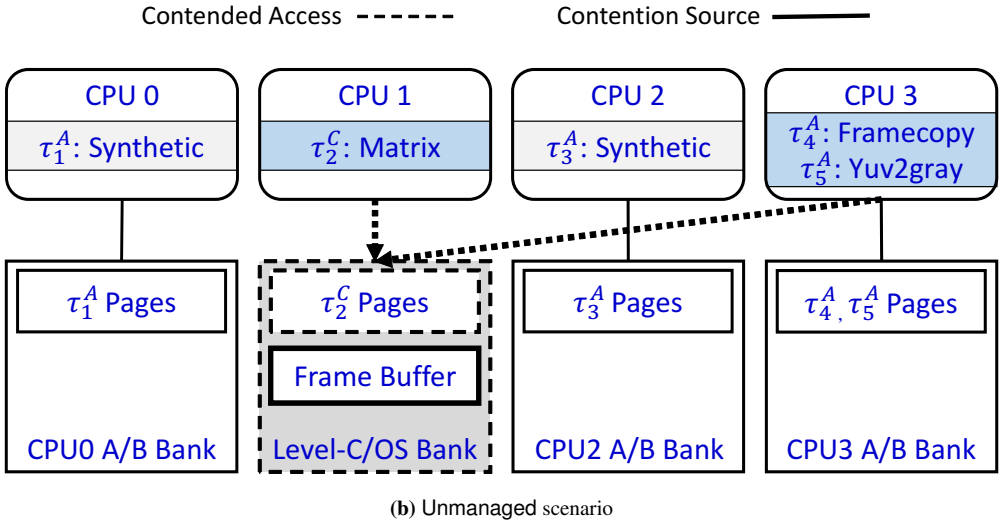


(b)

Figure 8 MOETs for **(a)** `load_msg()` of the Sender task and **(b)** `do_msg_fill()` of the Receiver task.



(a) Managed scenario



(b) Unmanaged scenario

Figure 9 Micro-benchmark tasks and resource allocations.

- Unmanaged: The frame buffer was instead allocated in a Level-C bank, as shown in Fig. 9(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. 10 presents normalized (relative to Managed) MOETs and ACETs 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 accessing the frame buffer. Fig. 12 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).

Observation 2 *CPU-sourced interference inflated the MOET of copying by up to 81% and of color conversion by up to 31%. It also inflated the respective ACETs by 85% and 25%.*

This observation is supported by insets (a) and (b) of Fig. 10 and confirms that DRAM interference due to I/O buffers can be problematic. The MOET inflation of 81% between Managed and Unmanaged is shown in the MOET of Framecopy.640x480, which is 5321.33 μs under Unmanaged and 2925.67 μs under Managed. We observed higher impact on the execution times of Framecopy because Yuv2gray requires additional computation to re-encode each pixel, and this computation is not affected by CPU-sourced interference. Fig. 11 shows the MOETs and ACETs of Framecopy and Yuv2gray. Note that the gap between Idle and Managed is caused by interference from unmanaged resources

(see Sec. 2). (This is not to be confused with the Unmanaged measurements, which were made while *no* resources were managed, included DRAM and cache.) Under Managed, two instances of Synthetic (τ_1^A and τ_3^A) contend for unmanaged resources, while under Idle, no competing workload exists.

Observation 3 *CPU-sourced interference inflated Matrix's MOET by 6% and its ACET by 3%.*

This observation is supported by insets (a) and (b) of Fig. 12. As expected, the execution-time inflation decreases as the number of LLC ways allocated to Matrix increases. 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. The smaller increases here (relative to those shown in Fig. 10), in addition to the smaller differences between average and maximum observed times, are due to the fact that Matrix is far more deterministic than the video-processing microbenchmarks, due to operating on a fixed set of inputs without communicating with other tasks or peripheral devices.

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 Synthetic task 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.[13]
- 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- and DMA-sourced interference.

Fig. 13 presents MOET data obtained for Synthetic under these two scenarios with a small (Fig. 13(a)) and large (Fig. 13(b)) LLC region allocated to it.

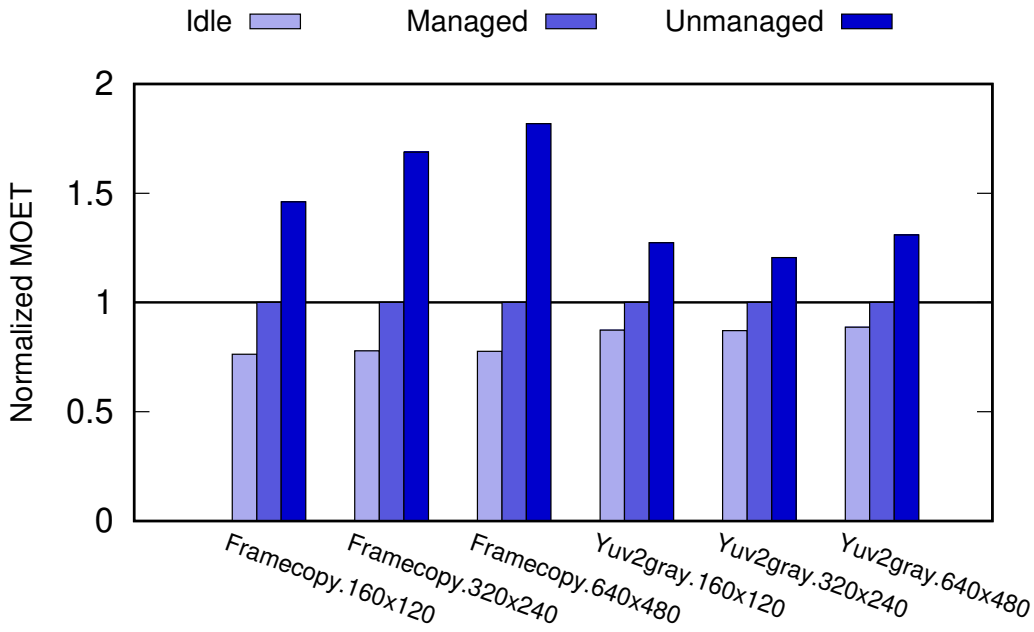
Observation 4 *Synthetic's MOET rose with increasing DMA bandwidth.*

This observation is supported by both insets of Fig. 13. Because this slowdown happened under both scenarios, we can infer that it is mainly due to memory-bus contention caused by DMA.

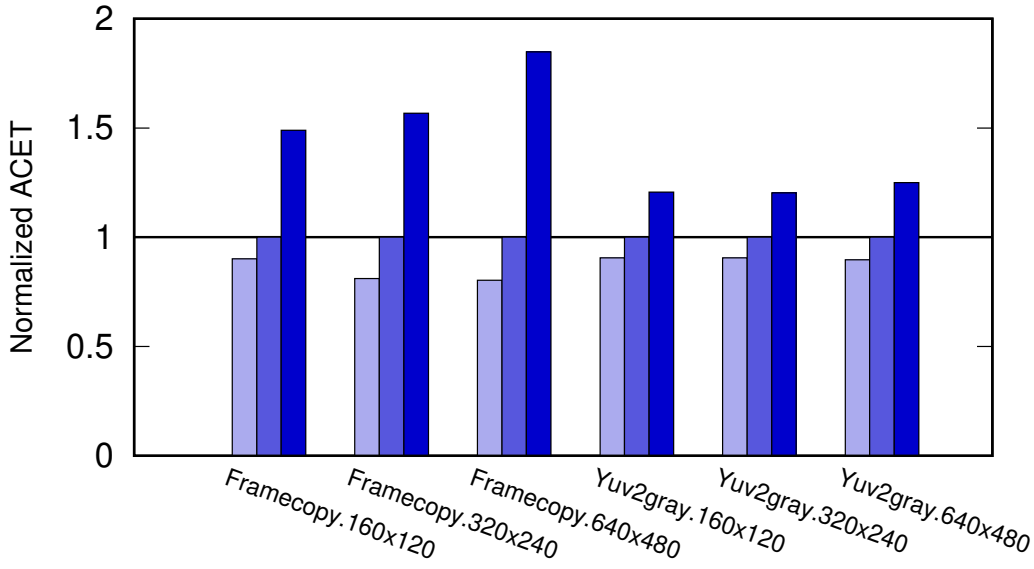
Observation 5 *Allocating an I/O buffer in a Level-A/B bank can have a negative impact on other Level-A/B tasks that access that bank.*

Fig. 13(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. 13(b).

[13] 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, and this experiment was only intended to isolate the impact of DMA-sourced interference on the measured (Synthetic) task.

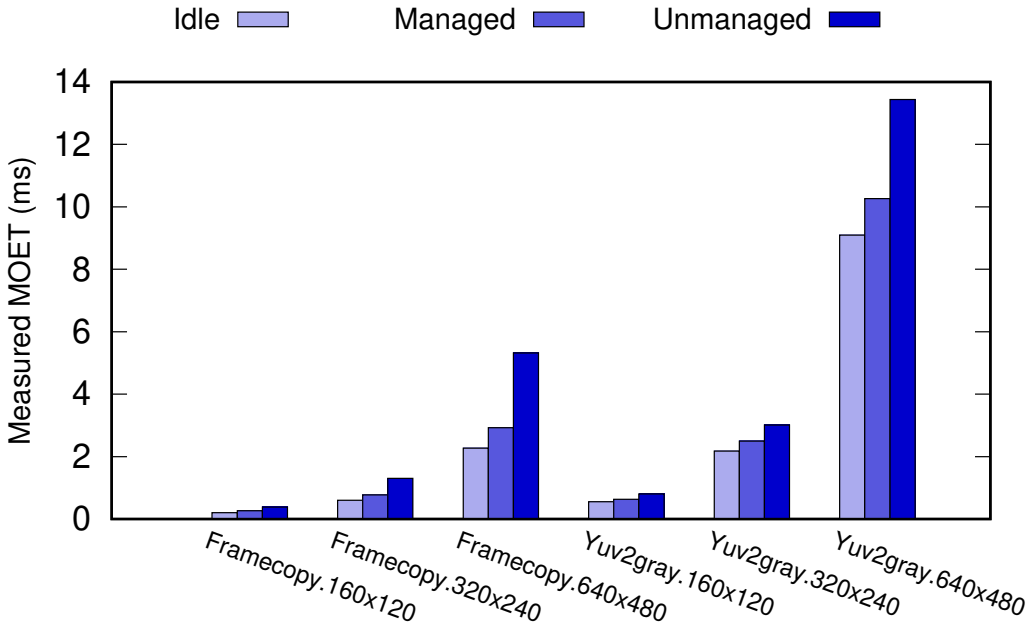


(a) MOETs

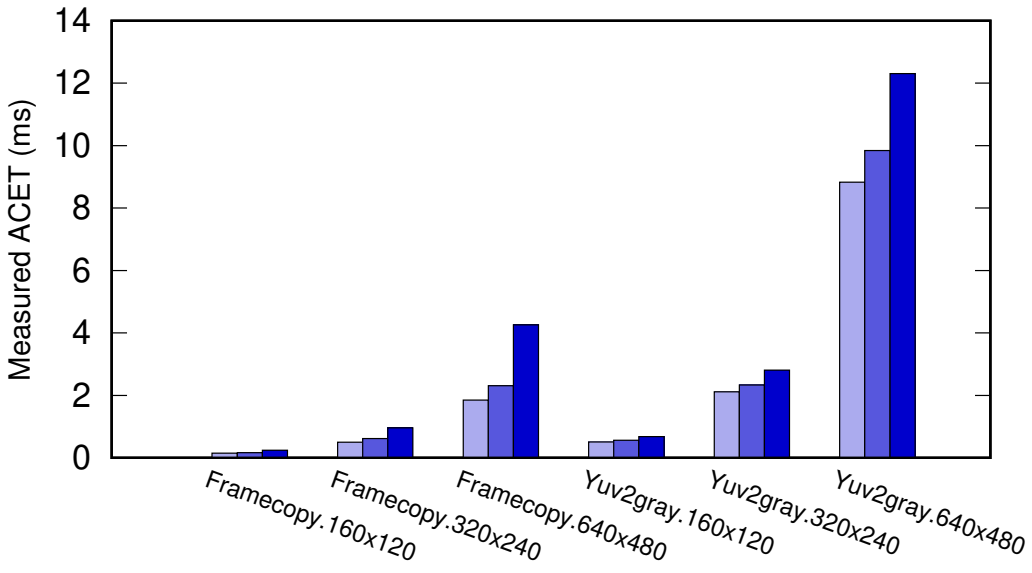


(b) ACETs

Figure 10 Normalized execution times of Framecopy and Yuv2gray.

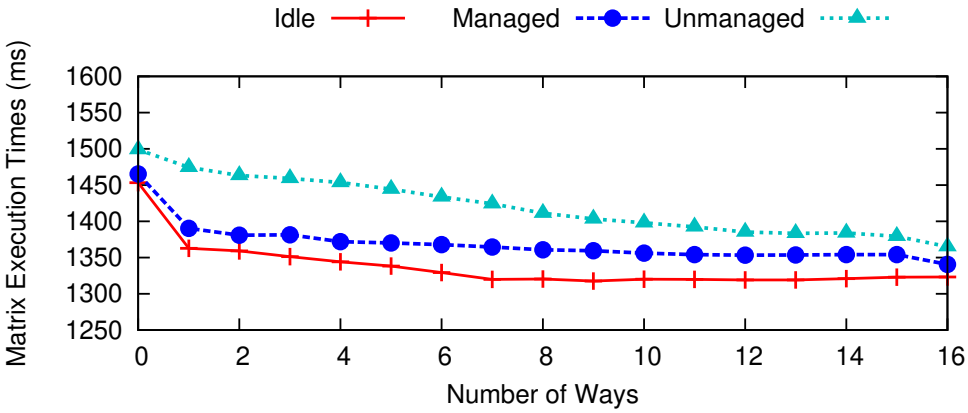


(a) MOETs

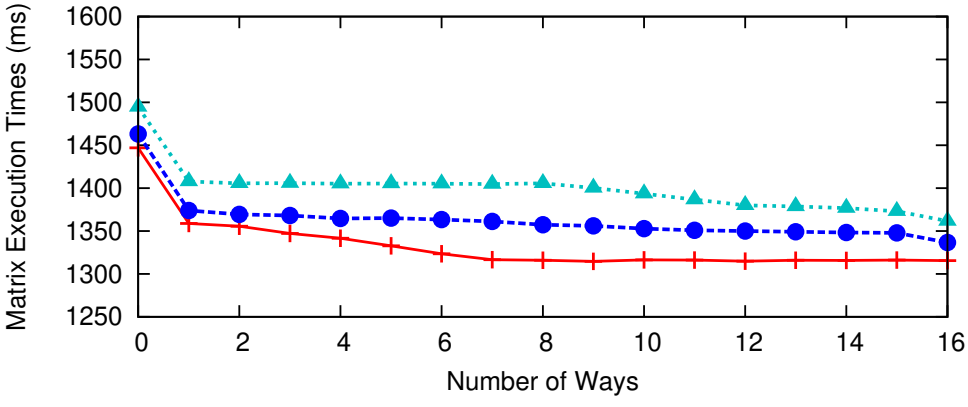


(b) ACETs

Figure 11 Measured execution times of Framecopy and Yuv2gray.

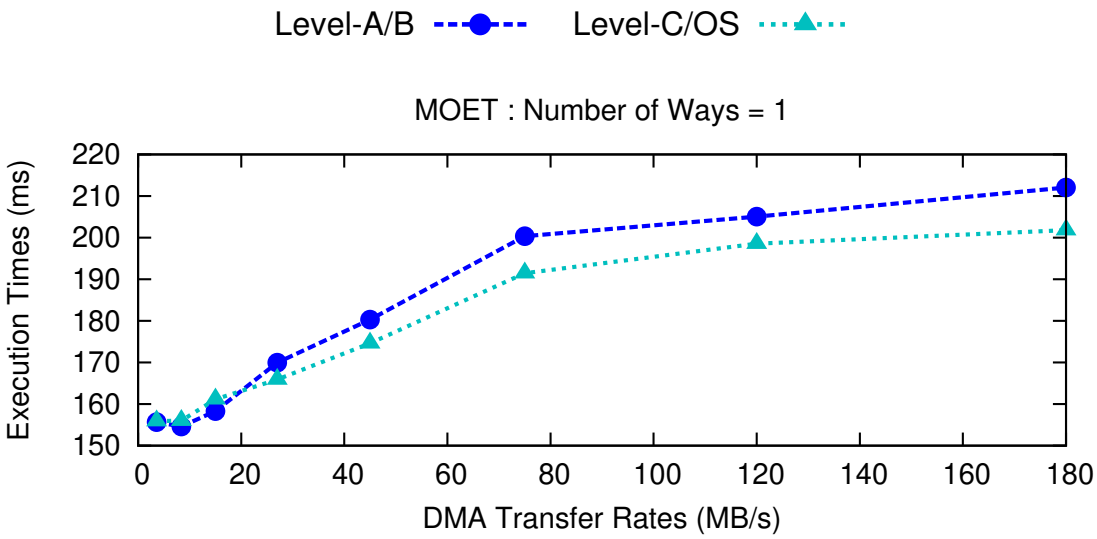


(a) MOETs

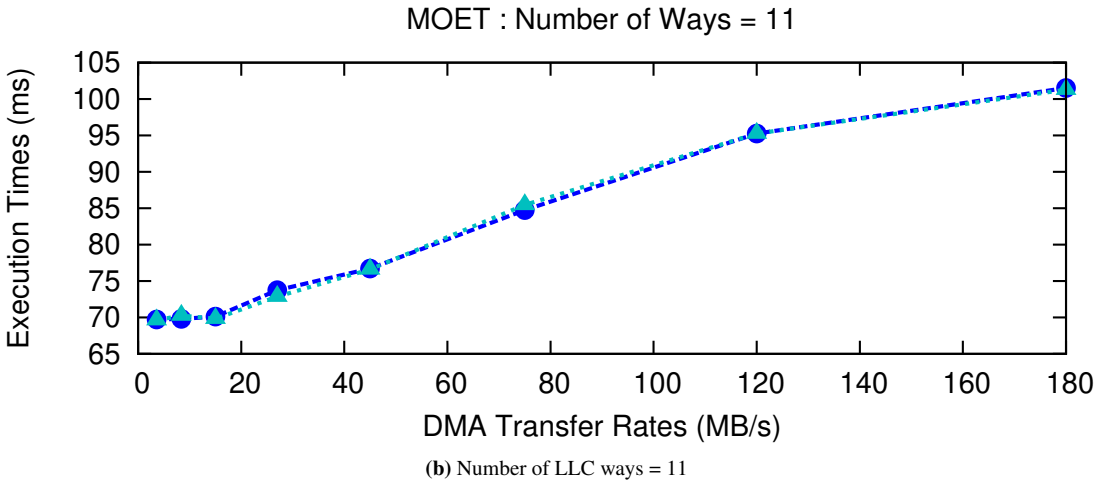


(b) ACETs

Figure 12 Matrix execution times as a function of allocated LLC space.



(a) Number of LLC ways = 1



(b) Number of LLC ways = 11

Figure 13 Synthetic MOETs under two allocation scenarios.

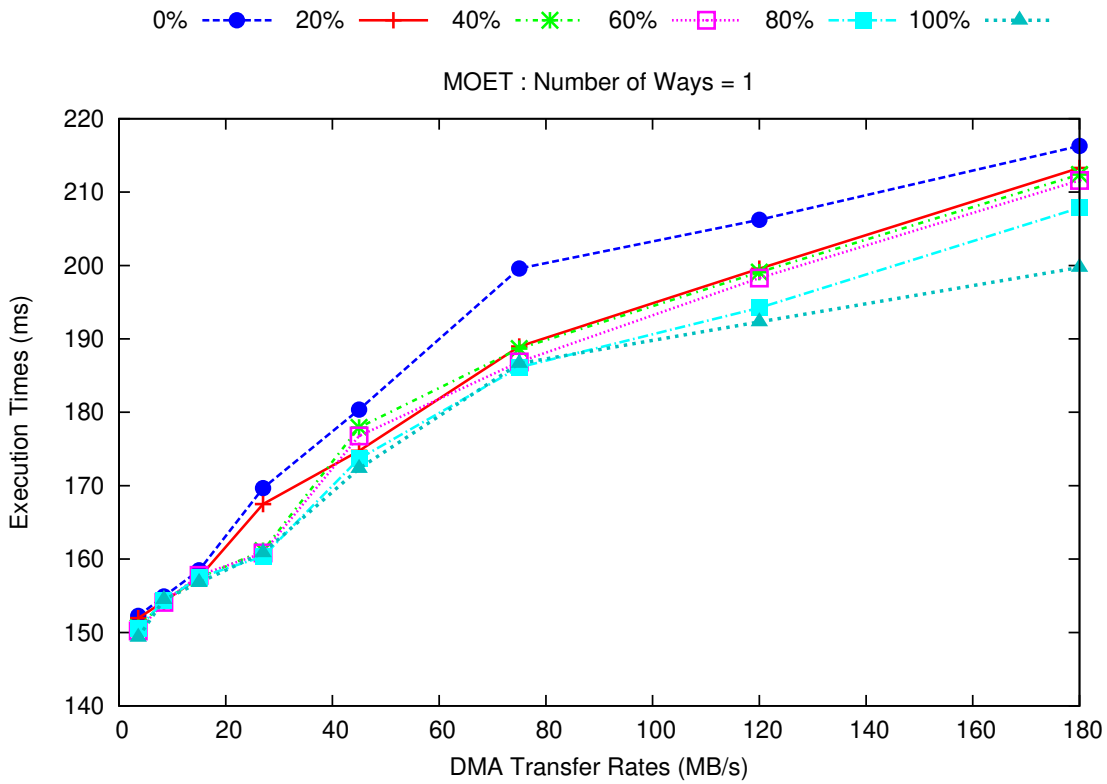


Figure 14 Synthetic MOETs under six different I/O-buffer allocations.

These results confirm in practice the tradeoff alluded to in Sec. 4.4. If an I/O-performing Level-A or -B task is allocated I/O buffers in its own Level-A/B bank, then the I/O-performing task does not experience CPU-sourced interference. However, the resulting DMA transfers can cause other tasks that access the same Level-A/B 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 the I/O-performing task experiences CPU-sourced interference when accessing the buffers. In either case, tasks executing on any CPU can experience interference from unmanaged resources as part of DMA-sourced interference. To further quantify this tradeoff, we conducted an additional micro-benchmark experiment.

Tradeoff between two allocation approaches. To evaluate the tradeoff between allocating buffers in Level-A/B or Level-C banks, we modified the Load-Generator task to read 400KB of data spread across both a Level-A/B bank and the Level-C banks. We evaluated six different scenarios, respectively allocating 0%, 20%, 40%, 60%, 80%, or 100% of the I/O buffers in the Level-C banks. For example, in the scenario where 40% of the I/O buffers are allocated in the Level-C banks, the Load-Generator task reads 160KB of data from the Level-C banks and 240KB of data from its core's Level-A/B bank.

Observation 6 *Synthetic's MOET decreases as the percentage of Level-C allocation increases.*

Fig. 14 supports this observation. A Level-A or -B non-I/O-performing task experiences more interference if it must share its Level-A/B bank with other tasks' I/O buffers. Such interference is caused by row-buffer conflicts in the Level-A/B bank as part of DMA-sourced interference. However, the I/O-performing tasks will experience less CPU-sourced interference. Therefore, the overall impact of CPU- and DMA-sourced interference depends on a ratio between non-I/O-performing tasks and I/O-performing tasks in a task system. 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 schedulability for entire task systems. To assess such tradeoffs, we conducted a large-scale, overhead-aware schedulability study.

Schedulability studies. Drawing concrete conclusions about different hardware-management techniques or real-time algorithms is difficult when the efficacy of each approach is highly dependent on task-system characteristics. For example, the additional I/O-management techniques introduced in this paper clearly would be of little use to a task system that performs no I/O. However, an I/O- or IPC-intensive task system may experience a large benefit from the additional management. Evaluating the efficacy of our hardware-management techniques using either one of these two extremes could result in a misleading conclusion, so we must instead evaluate our work with respect to a large number of task systems, while accounting for any new overheads that our techniques introduce.

Schedulability studies are a widely used tool in real-time research intended to address this difficult problem. Schedulability studies produce *generalizable* results about the utility of new techniques by generating the parameters for a large number of synthetic task systems (*i.e.*, not actual executable code), and then determining what fraction of the generated systems are schedulable using the techniques under study and an appropriate schedulability test. The claim then is that if a synthetic task system is schedulable, then a real task system with the same parameters would also be schedulable.

A large number of task systems is necessary in order to provide statistical confidence and to cover a broad spectrum of task-system parameters. The generated systems must be synthetic, consisting only of task parameters, *e.g.*, utilization, period, working set size, *etc.*, because generating and running executable code at the necessary scale is infeasible. However, schedulability studies still must model behavior (*i.e.*, interference) encountered on real hardware. To account for this, it is a common practice for schedulability studies to use microbenchmark experiments, as mentioned in Sec. 5, to model how the synthetically generated execution costs should be adjusted to represent the impact of interference and overheads (Brandenburg 2011).

6.1 Approach

We begin by describing the optimization framework used in this schedulability study.

Optimization framework. The overall objective of the optimization framework within this work is to lower the total utilization of Level-C tasks, as this generally lowers the tardiness of these tasks (we assume that the response times of Level-A/B tasks are irrelevant so long as they make their HRT deadlines). The framework explores a space of possible partitions of the hardware to accomplish this. However, the level of interference experienced by a task depends on how the hardware is partitioned, so the execution costs of tasks are conditioned on this partitioning. Thus, the framework is constrained to produce hardware partitionings under which the given task system passes all required schedulability tests.

The framework can be roughly divided into three phases. The first phase involves partitioning Level-A/B tasks among cores. The second phase involves deciding whether to allocate I/O buffers of Level-A/B tasks in Level-A/B or -C banks. The third phase involves partitioning the LLC ways. The ways are partitioned into six regions: four per-core regions for the exclusive access of Level-A/B tasks, a region for Level-C tasks and the OS, and a region for shared buffers locked into the LLC. While more optimal resource-management solutions might be found by considering these decisions together rather than in phases, the phases are kept separate in order to keep the framework tractable for a schedulability study.

The first phase is similar to bin packing. The underlying heuristic used to partition tasks is worst-fit, though this process is complicated by CE. CE produces an advantage when tasks that share buffers are partitioned onto the same core, but this has the consequence of making the combined tasks more difficult to pack efficiently. If CE is applied too aggressively, schedulability may even be lost for Level-A/B tasks if they cannot be packed into the four bins (cores). The packing heuristic used by this framework addresses this tradeoff by monitoring the remaining capacity of each core by applying CE less aggressively as capacity diminishes. This is done by initially packing groups of tasks that share buffers instead of packing them individually. If the remaining capacity becomes insufficient to pack these groups, the groups are split until either all tasks are successfully partitioned or the system is deemed unschedulable. Pseudocode of this packing heuristic is presented in App. B of Chisholm et al. (2016).

The second phase allocates all Level-A/B I/O buffers in Level-A/B banks. As stated in Sec. 4.3, this approach benefits Level-A/B tasks that consume the data in these buffers at the expense of Level-A/B tasks that share the same DRAM banks. While this approach may initially seem naïve, this decision is supported by observations that this generally increases the likelihood that a task system will be schedulable, which we will elaborate on later.

The third phase is solved by a mixed integer linear program (MILP). The decision variables of the MILP are the number of ways provided to each LLC partition. This MILP is largely similar to one explained in detail in prior work on MC^2 where the effects of sharing are not considered (Chisholm et al. 2015). The total utilization of Level-C tasks is reduced and schedulability is maintained by the MILP's objective function and constraints, respectively. Note that shrinking the per-core LLC partition will cause a higher rate of cache misses to the Level-A/B tasks on that core, thereby increasing their execution times. The same occurs with respect to Level-C tasks when shrinking the Level-C/OS LLC partition. Shrinking the region dedicated to locked buffers reduces the benefits of CL. The MILP balances these needs in the face of finite LLC space to produce an optimal LLC partitioning for the assignment decisions made in the first two phases.

Note that while we have mentioned CE and CL in this discussion of the optimization framework, we have not mentioned SBP. This is because the framework defaults to SBP for any buffers that it chooses not to apply CE or CL to due to the tradeoffs discussed above. This is necessary to avoid unpredictable cache evictions due to buffers, as discussed in Sec. 4.2. From the point of view of the optimization framework, Level-A/B tasks that access buffers under SBP have increased execution costs due to their accessing of Level-C DRAM banks, tasks that access buffers under CE have unchanged execution costs because the buffers are stored in the tasks' original Level-A/B banks, and tasks under CL have reduced execution costs because their buffers are permanently cached.

Schemes. In order to evaluate the individual strengths of the optimizations considered above, we considered nine 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 indicates how IPC buffers are handled, and y indicates how 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 described above.

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-LP (long periods in A/B banks): the I/O buffers belonging to the half (within each level) of the Level-A/B tasks with the longest periods are assigned to their corresponding tasks' Level-A/B banks, and others are assigned to Level-C banks;
- C+A/B-SP (short periods in A/B banks): the I/O buffers belonging to the half (within each level) of the Level-A/B tasks with the shortest periods are assigned to their corresponding tasks' Level-A/B banks, and others are assigned to Level-C banks;
- C+A/B-All (all A/B tasks in A/B banks): 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.

We considered nine different management schemes, taken from combinations of the choices listed above. The nine management schemes we considered are: R|R, C|C, O|C, C|C+A/B-ALL, C|C+A/B-LP, C|C+A/B-SP, O|C+A/B-ALL, O|C+A/B-LP, and O|C+A/B-SP. Note that O|C+A/B-ALL applies all optimizations as described in the optimization framework above. We considered R|R to illustrate the ill effects of paying no attention to OS buffer-assignment issues. C|C reflects the choice of coarse-grained partitioning in order to separate the OS from HRT tasks by simply preventing OS data structures from existing in Level-A/B banks entirely (a similar partitioning is possible on most RTOSs today). O|C, C|C+A/B-ALL, and O|C+A/B-ALL provide varying degrees of fine-grained partitioning in which certain OS data structures are allowed to exist in Level-A/B banks.

Note that the two choices of C and C+A/B-All 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 Level-A/B banks. While an optimal assignment of Level-A/B I/O buffers to banks would consider this entire spectrum of choices, this is not practical for a schedulability study that relies on solving a MILP, like our optimization framework. For example, allowing our optimization framework to make *per-buffer* choices, *i.e.*, allocate Level-A/B I/O buffers to Level-C or Level-A/B banks on a case-by-case basis, could be implemented by creating per-buffer binary decision variables that indicate whether each buffer is allocated to a Level-A/B bank. This would cause the number of integer variables in our MILP to scale with the number of I/O-performing tasks instead of the number of cores, explosively increasing our framework's runtime when applied to a large number of task systems.

Even though exploring the entire spectrum of choices is intractable, we consider the choices C+A/B-LP and C+A/B-SP in order to capture two midpoints between C and C+A/B-All. Unlike C and C+A/B-All, which mitigate one of either DMA- or CPU-sourced interference, C+A/B-LP and C+A/B-SP compromise between the two interference sources. These two choices allocate Level-A/B I/O buffers based on the length of their corresponding tasks' periods. Allocating by period is interesting because I/O-performing tasks with shorter periods request and access I/O data more frequently (we assume an I/O operation is performed during every job). Because these tasks perform I/O more frequently, they can potentially cause greater DMA-sourced interference, which may have negative effects on other high-criticality tasks if I/O buffers are allocated in Level-A/B banks. On the other hand, if these tasks' I/O buffers are located in Level-C banks, they will suffer greater CPU-sourced interference due to accessing these I/O buffers more frequently. Therefore, although both choices result in both CPU- and DMA-sourced interference, allocating the tasks with shorter periods in Levels-A/B (C+A/B-SP) or -C (C+A/B-LP) favors reducing CPU- and DMA-sourced interference, respectively.

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 complications due to precedence constraints, we assumed that tasks that communicate via IPC share a common period, like in prior work on user-level IPC in MC^2 (Chisholm et al. 2016).

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 (Chisholm et al. 2015, 2016; Kim et al. 2017a,b; Chisholm et al. 2017). 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 MOETs and ACETs, overheads pertaining to OS activities and I/O, cache reload times, *etc.* (we elaborate on details relevant to this paper below). The steps in the procedure are the following:

1. Choose distributions from the *first four* categories in Table 3.[14] 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* that is modified in the subsequent steps to introduce IPC and I/O.

2. Select distributions from Category 5 in Table 3. 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).

[14] 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) Level-A, -B, and -C execution costs model inflated worst-case, worst-case, and average-case execution costs, respectively. (4) is modeled based on measurement data. These details are described in full in previously published papers (Chisholm et al. 2016, 2015; Kim et al. 2017a,b).

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 3 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.

Note that the above procedure does not determine PETs. The value obtained by multiplying a task's generated period and utilization 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) under Level-A pessimism. Task execution costs under other management options and assumptions are determined by inflating this idle-system cost using micro-benchmark data pertaining to way allocations, I/O buffer allocations, and I/O bandwidths (data like that given in Figs. 4 and 13 is particularly relevant in our context). How a synthetic task's execution cost scales under different way allocations is modeled as an exponential decay function to reflect that allocating additional ways generally provide diminishing returns in terms of reducing the execution cost. Likewise, how a synthetic task's execution cost is inflated under different I/O bandwidths is modeled as a negative exponential decay function to reflect the behavior in Figs. 13 and 14 (that the increase in execution time caused by raising the I/O bandwidth decreases at higher bandwidths). The parameters of these scaling functions are sampled from ranges for each synthetic task to reflect that actual tasks would have different memory access patterns which would affect their response to their allocated LLC space or I/O bandwidth. These ranges are informed by our microbenchmark experiments. Buffer access times under CL, which dedicates some LLC space to buffers, are scaled similarly according to the size of the dedicated LLC region.

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. 8 for IPC and Fig. 10 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 the execution costs of Level-A/B tasks on CPU 0, where interrupts are handled. This is done by assuming the maximum possible number of interrupts delay the execution of every such job through its period. The sum of the processing times of these interrupts is added to the execution cost of every task. For Level-C tasks, we used interrupt-accounting techniques from Brandenburg et al. (2011), which are applicable under global scheduling.

As a cumulative example of how a task's conditional execution cost under a scheme may be calculated by inflating its idle-system cost, consider a hypothetical Level-A task that performs I/O under the C|C+A/B-SP scheme that

is scheduled on CPU 0. Assume that under C|C+A/B-SP, the I/O buffer corresponding to the considered task is stored in Level-C banks. Under this scheme, this task suffers interference from unmanaged resources, DMA-sourced interference due to peripherals accessing the considered task's Level-A/B bank, CPU-sourced interference due to other tasks accessing the DRAM bank containing the considered task's I/O buffer, and interrupts due to being scheduled on CPU 0 (*i.e.*, the CPU where all interrupts are directed—see Sec. 3). The degree of inflation due to unmanaged resources and DMA-sourced interference is informed by Fig. 14. The total DMA transfer rate and percentage of DMA directed at Level-A/B banks under C|C+A/B-SP for the task system the considered task belongs to are used to interpolate an execution time in Fig. 14. The inflation of this interpolated execution time relative to the MOET of an idle system (a DMA transfer rate of 0 MB/s) in Fig. 14 is used as a reference for how much to inflate the idle-system cost of the considered task. The inflation due to CPU-sourced interference is similarly informed by Fig. 10. Finally, to apply the interrupt accounting techniques mentioned previously, we took measurements of the execution times required to process these interrupts.

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 Table 3. 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.

6.2 Results

Our full study generated 648 schedulability plots, one per scenario, taking roughly 80 days of CPU time to compute. Our full set of plots is available online (Kim et al. 2019).

Fig. 15 shows three representative plots. The horizontal axis gives total system utilization.[15] 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. 15(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.

We now state several observations that follow from the full set of plots. We illustrate these observations using the plots in Fig. 15.

Observation 7 *Coarse I/O and IPC partitioning is beneficial.*

This observation is supported by the fact that the SUA of the C|C scheme is 5% larger than that of R|R. This can be observed qualitatively in Fig. 15, where the R|R curve is always below that for any other management scheme.

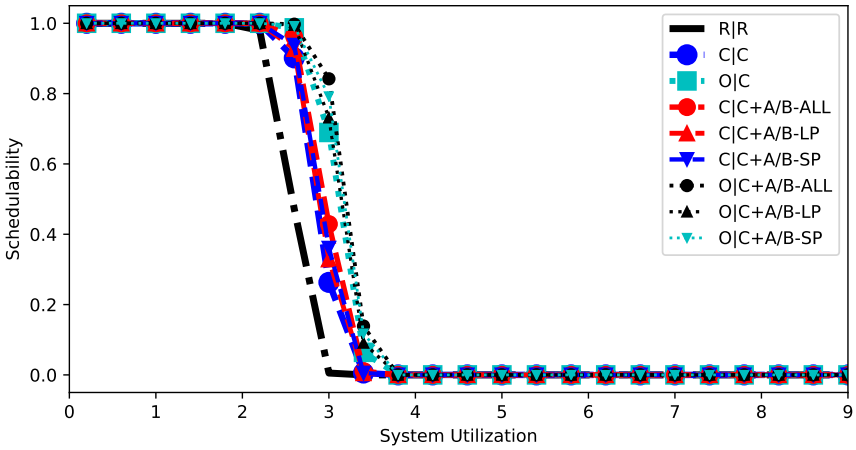
Observation 8 *IPC-buffer optimizations outperform coarse partitioning.*

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. 15. The 11% improvement increases to 13% when only *Large* (Table 3, Category 5) message sizes are considered. As expected, this indicates that the benefit is proportional to the amount of data shared with the OS.

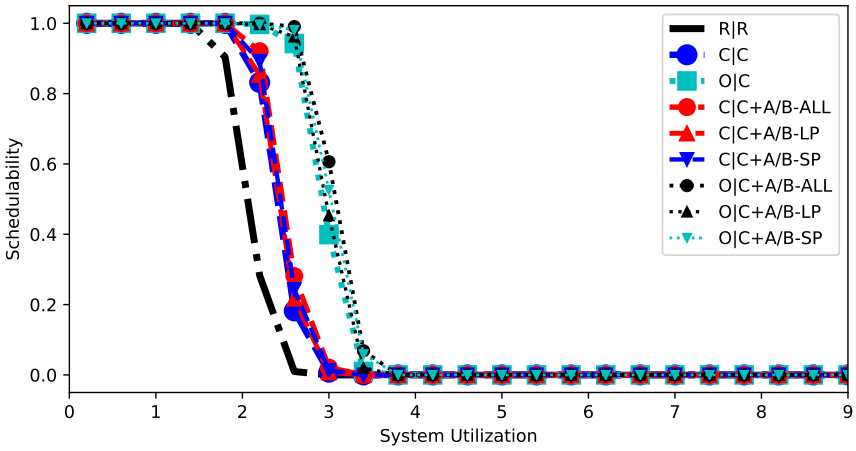
Observation 9 *Reducing CPU-sourced I/O interference is more important than reducing DMA-sourced interference.*

This observation addresses the tradeoff discussed in Sec. 4.3. The SUA of C|C+A/B-ALL 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 (Table 3, 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-ALL and C|C curves of insets (b) and (c) of Fig. 15.

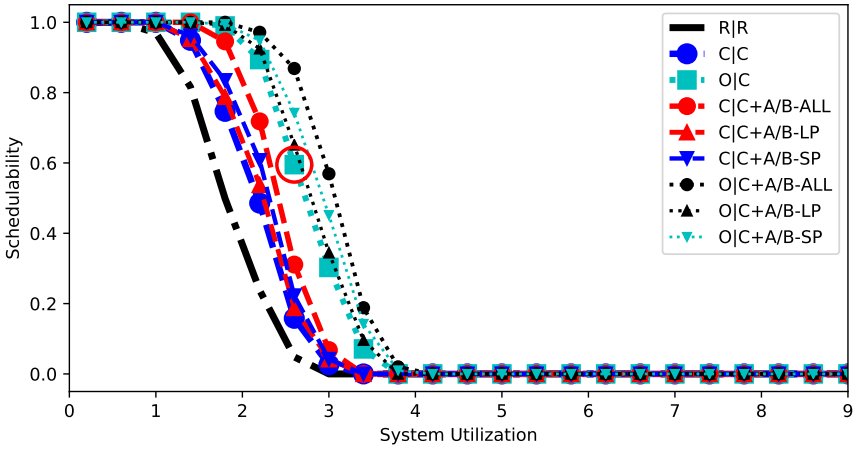
[15] The "utilization" referred to here is that initially obtained during task-set generation without accounting for MC^2's hardware management, which improves execution times. Thus, it is possible for a task system to have a total utilization exceeding four and be schedulable.



(a) (C-Light, Long, Light, Quick, Sm., High, Many)



(b) (C-Light, Long, Light, Quick, Lar., High, Many)



(c) (C-Light, Short, Med., Slow, Lar., High, Many)

Figure 15 Representative schedulability plots.

Observation 10 *No scheme that allocated some Level-A/B I/O buffers in Level-C outperformed allocating all such buffers in Level-A/B banks.*

This observation is supported by all insets of Fig. 15, where C|C+A/B-LP and C|C+A/B-SP cover at most the area of C|C+A/B-ALL. Quantitatively, the SUA of C|C+A/B-ALL is roughly 3% better than C|C+A/B-LP and C|C+A/B-SP. Likewise, O|C+A/B-LP and O|C+A/B-SP cover at most the area of O|C+A/B-ALL, and the SUA of O|C+A/B-ALL is 3% better than O|C+A/B-LP and 2% better than O|C+A/B-SP. In most scenarios, such as in insets (a) and (b) of Fig. 15, schedulability under C|C+A/B-LP and C|C+A/B-SP is much closer to the schedulability under C|C (coarse-grained OS isolation) than to C|C+A/B-ALL. Similarly, the SUA of O|C+A/B-LP and O|C+A/B-SP is closer to that of O|C (which only optimizes IPC buffer placement) than to O|C+A/B-ALL (which always puts high-criticality DMA buffers in Level-A/B banks). This suggests that schedulability is highly sensitive to CPU-sourced I/O interference, supporting Obs. 9.

Observation 11 *Used in conjunction, fine-grained I/O and IPC management outperform all other schemes.*

This observation is supported by Fig. 15, where O|C+A/B-ALL covers the greatest area in all insets. Quantitatively, the SUA of O|C+A/B-ALL 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-ALL exhibits a 20% 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. 15(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 (Huang et al. 2006, 1996a,b), implementing DMA schedulers using hardware mediation (Huang et al. 2003; Pellizzoni et al. 2008a; Pellizzoni and Caccamo 2010; Pellizzoni et al. 2010), mixed-criticality systems (Muench et al. 2014), and scheduling legacy I/O operations (Kim et al. 2014b). 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 *spatial* isolation.

Other research has focused on reducing DRAM-bank and cache contention via memory-allocator improvements (Yun et al. 2014; Liao et al. 2017). Of these, PALLOC (Yun et al. 2014), 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 (Kotaba et al. 2013). Prior efforts have focused on issues such as cache partitioning (Altmeyer et al. 2014; Herter et al. 2011; Kim et al. 2013; Scolari et al. 2016; Xu et al. 2017, 2016; Awan et al. 2017), DRAM controllers (Audsley 2013; Guo and Pellizzoni 2017; Jalle et al. 2014; Kim et al. 2015; Krishnapillai et al. 2014; Muralidhara et al. 2011), and bus-access control (Alhammad and Pellizzoni 2016; Alhammad et al. 2015; Giannopoulou et al. 2013; Hassan and Patel 2016; Hassan et al. 2015; Pellizzoni et al. 2010). Other work has focused on reducing shared-resource interference when per-core scratchpad memories are used (Tabish et al. 2016b), throttling lower-criticality tasks' memory accesses (Yun et al. 2012), and controlling bandwidth allocations (Seetanadi et al. 2017).

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.

This paper builds upon and extends the prior conference paper (Kim et al. 2018). The main additions are: (i) we have provided further implementation details regarding the memory-management system; (ii) we added additional micro-benchmark experiments; (iii) we considered finer-grained I/O-buffer allocation heuristics in Sec. 6.

References

Alhammad A, Pellizzoni R (2016) Trading cores for memory bandwidth in real-time systems. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 1–11

Alhammad A, Wasly S, Pellizzoni R (2015) Memory efficient global scheduling of real-time tasks. In: Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, pp 285–296

Altmeyer S, Douma R, Lunniss W, Davis R (2014) Evaluation of cache partitioning for hard real-time systems. In: Proceedings of the 26th Euromicro Conference on Real-Time Systems, pp 15–26

ARM Limited (2009) Application Note 228: Implementing DMA on ARM SMP Systems. `http://infocenter.arm.com/help/topic/com.arm.doc.dai0228a/DAI228A_DMA_on_SMP_systems.pdf`

ARM Limited (2010) AMBA® Network Interconnect (NIC-301): Technical Reference Manual. `https://static.docs.arm.com/ddi0397/g/DDI0397G_amba_network_interconnect_nic301_r2p1_trm.pdf`

Audsley N (2013) Memory architecture for NoC-based real-time mixed criticality systems. In: Proceedings of the 1st International Workshop on Mixed Criticality Systems

Awan MA, Bletsas K, Souto P, Akesson B, Tovar E (2017) Mixed-criticality scheduling with dynamic redistribution of shared cache. In: Proceedings of the 29th Euromicro Conference on Real-Time Systems, pp 18:1–18:21

Brandenburg B (2011) Scheduling and locking in multiprocessor real-time operating systems. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC

Brandenburg B, Leontyev H, Anderson J (2011) An overview of interrupt accounting techniques for multiprocessor real-time systems. Journal of Systems Architecture 57(6):638–654

Burns A, Davis R (2019) Mixed criticality systems – A review. Tech. rep., Department of Computer Science, University of York

Certification Authorities Software Team (2016) Position paper CAST-32A: Multi-core processors

Chisholm M, Ward B, Kim N, Anderson J (2015) Cache sharing and isolation tradeoffs in multicore mixed-criticality systems. In: Proceedings of the 36th IEEE International Real-Time Systems Symposium, pp 305–316

Chisholm M, Kim N, Ward B, Otterness N, Anderson J, Smith FD (2016) Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In: Proceedings of the 37th IEEE International Real-Time Systems Symposium, pp 57–68

Chisholm M, Kim N, Tang S, Otterness N, Anderson J, Smith F, Porter D (2017) Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. In: Proceedings of the 25th International Conference on Real-Time Networks and Systems, pp 58–67

Erickson J, Kim N, Anderson J (2015) Recovering from overload in multicore mixed-criticality systems. In: Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium, pp 775–785

Freescale (2014) i.MX 6Dual/6Quad Applications Processor Reference Manual. https://www.nxp.com/webapp/Download?colCode=IMX6DQRM

Giannopoulou G, Stoimenov N, Huang P, Thiele L (2013) Scheduling of mixed-criticality applications on resource-sharing multicore systems. In: Proceedings of the 13th International Conference on Embedded Software, pp 1–15

Gorman M (2004) Describing physical memory. In: Understanding the Linux Virtual Memory Manager, Prentice Hall PTR, Upper Saddle River, NJ, USA, chap 2

Guo D, Pellizzoni R (2017) A requests bundling DRAM controller for mixed-criticality systems. In: Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 247–258

Hassan M, Patel H (2016) Criticality- and requirement-aware bus arbitration for multi-core mixed criticality systems. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 1–11

Hassan M, Patel H, Pellizzoni R (2015) A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In: Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, pp 307–316

Herman J, Kenna C, Mollison M, Anderson J, Johnson D (2012) RTOS support for multicore mixed-criticality systems. In: Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 197–208

Herter J, Backes P, Haupenthal F, Reineke J (2011) CAMA: A predictable cache-aware memory allocator. In: Proceedings of the 23rd Euromicro Conference on Real-Time Systems, pp 23–32

Huang TY, Liu JWS, Chung JY (1996a) Allowing cycle-stealing direct memory access I/O concurrent with hard-real-time programs. In: Proceedings of the 4th International Conference on Parallel and Distributed Systems, pp 422–429

Huang TY, Liu JWS, Hull D (1996b) A method for bounding the effect of DMA I/O interference on program execution time. In: Proceedings of the 17th IEEE Real-Time Systems Symposium, pp 275–285

Huang TY, Chou CC, Chen PY (2003) Bounding the execution times of DMA I/O tasks on hard-real-time embedded systems. In: Proceedings of the 9th International Conference on Real-Time and Embedded Computer Systems and Applications, pp 499–512

Huang TY, Chou CC, Chen PY (2006) Bounding DMA interference on hard-real-time embedded systems. Journal of Information Science and Engineering 22:1229–1247

Jalle J, Quinones E, Abella J, Fossati L, Zulianello M, Cazorla P (2014) A dual-criticality memory controller (DCmc) proposal and evaluation of a space case study. In: Proceedings of the 35th IEEE Real-Time Systems Symposium, pp 207–217

Kim H, Kandhalu A, Rajkumar R (2013) A coordinated approach for practical OS-level cache management in multi-core real-time systems. In: Proceedings of the 25th Euromicro Conference on Real-Time Systems, pp 80–89

Kim H, de Niz D, Andersson B, Klein M, Mutlu O, Rajkumar R (2014a) Bounding memory interference delay in COTS-based multi-core systems. In: Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 145–154

Kim H, Broman D, Lee E, Zimmer M, Shrivastava A, Oh J (2015) A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In: Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium, pp 317–326

Kim J, Yoon M, Bradford R, Sha L (2014b) Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems. In: Proceedings of the 38th IEEE Annual Computer, Software, and Applications Conference, pp 321–331

Kim N (2019) Combining hardware management with mixed-criticality provisioning in multicore real-time systems. PhD thesis, University of North Carolina at Chapel Hill, Chapel Hill, NC, available at http://www.cs.unc.edu/~anderson/diss/namhoondiss.pdf

Kim N, Chisholm M, Otterness N, Anderson J, Smith FD (2017a) Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In: Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 223–234

Kim N, Ward B, Chisholm M, Anderson J, Smith FD (2017b) Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. Real-Time Systems 53(5):709–759

Kim N, Tang S, Otterness N, Anderson J, Smith FD, Porter D (2018) Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks. In: Proceedings of the 26th International Conference on Real-Time Networks and Systems, pp 191–201

Kim N, Tang S, Otterness N, Anderson J, Smith F, Porter D (2019) Supporting I/O and IPC via fine-grained OS isolation for mixed-criticality real-time tasks. Full version of this paper, available at http://www.cs.unc.edu/~anderson/papers.html

Knowlton K (1965) A fast storage allocator. Communications of the ACM 8(10):623–624

Knuth D (1968) The Art of Computer Programming. Addison-Wesley

Kotaba O, Nowotsch J, Paulitsch M, Petters S, Theiling H (2013) Multicore in real-time systems – temporal isolation challenges due to shared resources. In: Proceedings of the Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems, pp 1–6

Krishnapillai Y, Wu Z, Pellizzoni R (2014) ROC: A rank-switching, open-row DRAM controller for time-predictable systems. In: Proceedings of the 26th Euromicro Conference on Real-Time Systems, pp 27–38

Liao X, Guo R, Jin H, Yue J, Tan G (2017) Enhancing the malloc system with pollution awareness for better cache performance. IEEE Transactions on Parallel and Distributed Systems 28(3):731–745

LITMUS$^{\mathrm{RT}}$ Project (2018) LITMUS$^{\mathrm{RT}}$: Linux testbed for multiprocessor scheduling in real-time systems. http://www.litmus-rt.org/

Liu L, Cui Z, Xing M, Bao Y, Chen M, Wu C (2012) A software memory partition approach for eliminating bank-level interference in multicore systems. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pp 367–376

Mollison M, Erickson J, Anderson J, Baruah S, Scoredos J (2010) Mixed criticality real-time scheduling for multicore systems. In: Proceedings of the 7th IEEE International Conference on Computer and Information Technology, pp 1864–1871

Muench D, Paulitsch M, Herkersdorf A (2014) Temporal separation for hardware-based I/O virtualization for mixed-criticality embedded real-time systems using PCIe SR-IOV. In: Proceedings of the Workshop on Architecture of Computing Systems, pp 1–7

Muralidhara S, Subramanian L, Mutlu O, Kandemir M, Moscibroda T (2011) Reducing memory interference in multicore systems via application-aware memory channel partitioning. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp 374–385

Musmanno J (2003) Data intensive systems (DIS) benchmark performance summary

Pellizzoni R, Caccamo M (2010) Impact of peripheral-processor interference on WCET analysis of real-time embedded systems. IEEE Transactions on Computers 59(3):400–415

Pellizzoni R, Bui B, Caccamo M (2008a) Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In: Proceedings of the 29th IEEE Real-Time Systems Symposium, pp 221–231

Pellizzoni R, Bui B, Caccamo M, Sha L (2008b) Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In: Proceedings of the 29th IEEE Real-Time Systems Symposium, pp 221–231

Pellizzoni R, Schranzhofer A, Chen J, Caccamo M, Thiele L (2010) Worst case delay analysis for memory interference in multicore systems. In: Proceedings of the Design, Automation Test in Europe Conference Exhibition, pp 741–746

Scolari A, Bartolini DB, Santambrogio MD (2016) A software cache partitioning system for hash-based caches. ACM Transactions on Architecture and Code Optimization 13(4):1–24

Seetanadi G, Camara J, Almeida L, Arzen K, Maggio M (2017) Event-driven bandwidth allocation with formal guarantees for camera networks. In: Proceedings of the 38th IEEE Real-Time Systems Symposium, pp 243–254

Tabish R, Mancuso R, Wasly S, Alhammad A, Phatak S, Pellizzoni R, Caccamo M (2016a) A real-time scratchpad-centric OS for multi-core embedded systems. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 1–11

Tabish R, Mancuso R, Wasly S, Alhammad A, Phatak S, Pellizzoni R, Caccamo M (2016b) A real-time scratchpad-centric OS for multi-core embedded systems. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 1–11

Valsan P, Yun H, Farshchi F (2016) Taming non-blocking caches to improve isolation in multicore real-time systems. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 1–12

Vestal S (2007) Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium, pp 239–243

Ward B, Herman J, Kenna C, Anderson J (2013) Making shared caches more predictable on multicore platforms. In: Proceedings of the 25th Euromicro Conference on Real-Time Systems, pp 157–167

Xu M, Phan LTX, Choi HY, Lee I (2016) Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 1–12

Xu M, Phan LTX, Choi HY, Lee I (2017) vCAT: Dynamic cache management using CAT virtualization. In: Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 211–222

Yun H, Yao G, Pellizzoni R, Caccamo M, Sha L (2012) Memory access control in multiprocessor for real-time systems with mixed criticality. In: Proceedings of the 24th Euromicro Conference on Real-Time Systems, pp 299–308

Yun H, Mancuso R, Wu Z, Pellizzoni R (2014) PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In: Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 155–166