Noname manuscript No.
(will be inserted by the editor)

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

Namhoon Kim · Bryan C. Ward · Micaiah Chisholm · James H. Anderson · F. Donelson Smith

Abstract *The multicore revolution is having limited impact in safety-critical application domains. A key reason is the "one-out-of-m" problem: when validating real-time constraints on an m-core platform, excessive analysis pessimism can effectively negate the processing capacity of the additional $m-1$ cores so that only "one core's worth" of capacity is utilized even though m cores are available. Two approaches have been investigated previously to address this problem: mixed-criticality allocation techniques, which provision less-critical software components less pessimistically, and hardware-management techniques, which make the underlying platform itself more predictable. A better way forward may be to combine both approaches, but to show this, fundamentally new criticality-cognizant hardware-management tradeoffs must be explored. Such tradeoffs are investigated herein in the context of a new variant of a mixed-criticality framework, called MC^2, that supports configurable criticality-based hardware management. This framework allows specific DRAM memory banks and areas of the last-level cache (LLC) to be allocated to certain groups of tasks. A linear-programming-based optimization framework is presented for sizing such LLC areas, subject to conditions for ensuring MC^2 schedulability. The effectiveness of the overall framework in resolving hardware-management and scheduling tradeoffs is investigated in the context of a large-scale overhead-aware schedulability study. This study was guided by extensive trace data obtained by executing benchmark programs on the new variant of MC^2 presented herein. This study shows that mixed-criticality allocation and hardware-management techniques can be much more effective when applied together instead of alone.*

Keywords *Hard real-time · Hardware management · Mixed-criticality · Multicore systems · Soft real-time*

Work supported by U.S. National Science Foundation grants CNS 1115284, CNS 1218693, CPS 1239135, CNS 1409175, and CPS 1446631, U.S. Air Force Office of Scientific Research grant FA9550-14-1-0161, U.S. Army Research Office grant W911NF-14-1-0499, and a grant from General Motors. The second author was also supported by an U.S. National Science Foundation graduate fellowship.

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

1 Introduction

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

The root of this problem is that shared hardware resources, such as caches, buses, and memory banks, are not predictably managed. As reviewed later, several proposals for predictably managing such resources have been presented that strive to reduce pessimism by enabling tighter task execution-time estimates. While these approaches seem promising, another way forward is the application of *mixed-criticality* (*MC*) analysis assumptions, as originally proposed by Vestal (2007). Under such assumptions, less-critical tasks are provisioned somewhat optimistically, allowing for increased platform utilization. These research directions share a similar goal of improving platform utilization, but are themselves orthogonal, raising broader research questions pertaining to the combination of both approaches. Can better platform utilization be realized if resources are managed differently at different criticality levels? If so, how should resources be managed both within and across criticality levels?

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

In this paper, we report on our efforts to construct an experimental platform that enables such tradeoffs to be assessed, and discuss the results of an experimental investigation conducted to provide such an assessment. Our new platform extends a framework called MC^2 (mixed-criticality on multicore) (Herman et al. 2012; Mollison et al. 2010; Ward et al. 2013), which has been the subject of continuing research by our group, by adding support for several hardware-management techniques. Specifically, we provide management that allows specific DRAM memory banks and areas of the last-level cache (LLC) to be assigned to certain groups of tasks; in the allocations considered herein, such task groups are determined in a criticality-cognizant way. Additionally, we provide techniques that isolate the operating system (OS) from user-space tasks with respect to the LLC and DRAM banks; to our knowledge, the issue of

[1] Multicore-related certification difficulties are extensively discussed in a recent position paper from the U.S. Federal Aviation Administration (Certification Authorities Software Team (CAST) 2014).

OS isolation has not been considered before in work on hardware management. To enable LLC areas to be sized appropriately, we also present an optimization framework that determines such areas subject to MC^2 schedulability conditions. We regard MC^2 as a rich and interesting platform for our investigation because (as discussed later) it supports several criticality levels (not just two, as typically assumed in work on MC scheduling), has both hard real-time (HRT) and soft real-time (SRT) components, both priority-scheduled and time-triggered components, and both partitioned and globally scheduled components.

Contributions. Our contributions are fourfold. First, after providing needed background (Sec. 2), we describe the hardware-management mechanisms we added to MC^2 (Sec. 3). The resulting MC^2 variant is highly configurable and breaks new ground by allowing sharing and isolation tradeoffs to be studied in a criticality-cognizant way.

Second, we discuss the results of micro-benchmark experiments conducted to assess such tradeoffs (Sec. 4). In these experiments, the impact of strong hardware isolation on task executions times is compared to the impact of permissive hardware sharing for tasks of both high criticality and low criticality.

Third, we provide techniques for optimizing the allocation of LLC areas in the context of our new MC^2 variant (Sec. 5). In particular, based on the results of the micro-benchmark experiments just mentioned, we propose a formal model in which the impacts of different allocation strategies are factored into the determination of LLC-related overheads and task execution times. Based on this model, we then present a linear program (LP) that determines LLC allocations that are beneficial for MC^2 schedulability.

Fourth, we provide evidence in favor of combining MC analysis with hardware management in attacking the one-out-of-m problem. This evidence is provided in the form of a large-scale overhead-aware schedulability study (Sec. 6.1) that we conducted to demonstrate the benefits of combining both approaches, and associated runtime experiments that we conducted to partially assess the reasonableness of some of the assumptions underlying this study (Sec. 6.2). To our knowledge, this is the first overhead-aware schedulability study that considers MC scheduling, hardware management, and a combination of both.

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

2 Background

We begin by reviewing needed background and related work.

Task model. We consider real-time workloads specified using the implicit-deadline *periodic task model* and assume familiarity with this model. We specifically consider

a task system $\tau = \{\tau_1, \ldots, \tau_n\}$, scheduled on m processors,[2] where task τ_i's *period* and *worst-case execution time* (*WCET*) are denoted T_i and e_i, respectively.[3] (We generalize this model below when considering MC scheduling.) The *utilization* of task τ_i is given by $u_i = e_i/T_i$ and the *total system utilization* is defined as $\sum_i u_i$. A periodic task system may be scheduled following a *partitioned scheduling* approach (tasks are statically assigned to processors), a *global scheduling* approach (any task may execute on any processor), or some hybrid of the two. If a job of τ_i has a deadline at time d and completes execution at time t, then its *tardiness* is $max\{0, t - d\}$. Tardiness should be zero for any job of a HRT task, and should be bounded by a (reasonably small) constant for any job of a SRT task.

Mixed-criticality scheduling. The roots of most recent work on MC scheduling can be traced to a seminal paper by Vestal (2007). For systems where tasks of differing criticalities exist, he proposed adopting less-pessimistic execution-time assumptions when checking the schedulability of less-critical tasks. More formally, in a system with L criticality levels, each task has a *provisioned execution time* (*PET*)[4] specified at every level, and L system variants are analyzed: in the Level-ℓ variant, the real-time requirements of all Level-ℓ tasks are verified with Level-ℓ PETs assumed for *all* tasks (at any level). The degree of pessimism in determining PETs is level-dependent: if Level ℓ is of higher criticality than Level ℓ', then Level-ℓ PETs will generally be greater than Level-ℓ' PETs. For example, in the systems considered by Vestal (2007), observed WCETs were used to determine PETs for tasks at lower levels, and such times were inflated to determine PETs at higher levels. The task model resulting from Vestal's work has come to be known as the *MC task model*.

MC^2. Vestal's work led to a significant body of follow-up work (see Burns and Davis (2016) for an excellent survey). Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors (to our knowledge) (Herman et al. 2012; Mollison et al. 2010; Ward et al. 2013). MC^2 is implemented as a LITMUS$^{\mathrm{RT}}$ plugin[5] and supports four criticality levels, denoted A (highest) through D (lowest), as shown in Fig. 1. Higher-criticality tasks are statically prioritized over lower-criticality ones. Level-A tasks are partitioned and scheduled

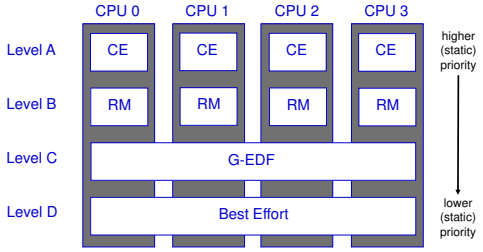


Fig. 1 Scheduling in MC^2 on a quad-core machine.

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

[3] The notation C_i is now commonly used to denote a task execution time, but the term "C" has a pre-existing meaning in the context of MC^2.

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

[5] LITMUS$^{\mathrm{RT}}$ is a real-time extension of the Linux kernel. Source code is available at `http://www.litmus-rt.org`.

on each core using a time-triggered table-driven cyclic executive.[6] Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each core.[6] On each core, the Level-A and -B tasks are required to be simply periodic (all tasks commence execution at time 0 and periods are harmonic), with the Level-B task periods being integer multiples of the Level-A hyper-period. Level-C tasks are scheduled via a global earliest-deadline-first (GEDF) scheduler.[6] Level-A and -B tasks are HRT, Level-C tasks are SRT, and Level-D tasks are non-real-time; in this work, we assume that Level D is not present. MC^2 is a flexible framework. For example, it can be configured to have only two HRT criticality levels (as in most theoretical work on MC scheduling) or to fully assign the Level-A and -B subsystems to distinct, dedicated cores.

We adopt a measurement-based approach to determining PETs because work on static timing-analysis tools for multicore machines has not matured to the point of being directly applicable. Moreover, measurement-based processes for determining PETs are often used in practice. As in other prior work on MC^2 (Herman et al. 2012; Mollison et al. 2010; Ward et al. 2013), we assume that Level-C PETs reflect measured average-case execution times[7] (since Level C is SRT) and that Level-B PETs reflect measured worst-case execution times (since Level B is HRT). Further, we assume that Level-A PETs are defined by applying an inflation factor to Level-B PETs (since Level A is of highest criticality).

In MC^2 as implemented in LITMUS$^{\mathrm{RT}}$, a task's PET at its own criticality level is treated as an OS-enforced execution budget. If a job of a Level-ℓ task τ_i has an execution time exceeding τ_i's Level-ℓ budget, then more than one budget allocation will be required to service it. Note that a Level-A job's budget may be *sliced*. Under the cyclic-executive model (Baker and Shaw 1988), scheduling is based on fixed-length *frames*. Each Level-A job runs non-preemptively within a frame unless it is sliced. Job slicing allocates different portions of a job (job slices) to different frames.

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

Page coloring. One of the mechanisms applied in our new variant of MC^2 to manage hardware is an old technique called page coloring, which can be applied to eliminate interference within both the LLC and memory banks (Kessler and Hill 1992). We explain the basic idea here with respect to the LLC (which we assume to be set-associative). Consider the pages of physical memory in turn. Assign the color "0" to the first page, and assign the same color to the sets in the LLC to which its content's addresses map. In a similar way, assign the color "1" to the next page

[6] Other per-level schedulers optionally can be used, and Level-C tasks can be defined according to the sporadic task model. These options, and other considerations, such as slack reallocation, are discussed in prior papers (Herman et al. 2012; Mollison et al. 2010; Ward et al. 2013).

[7] As explained in Mills and Anderson (2011), tardiness bounds with respect to deterministic budget allocations at Level C can be used to bound tardiness in expectation when average-case task execution times are assumed.

and corresponding cache sets, and so on. Eventually, such color assignments will "wrap" and we will sequence through the same colors again. This process ensures that differently colored pages map to different sets in the LLC. Thus, accesses to two differently colored pages cannot cause cache conflicts. Note that this coloring process is based on physical memory addresses. Such addresses also determine how memory pages map to DRAM banks, so pages can also be colored with respect to the banks to which they are mapped.

Ensuring isolation with respect to the LLC. In most prior work on eliminating or reducing interference in the LLC, some variant of cache partitioning is used (see Kirk (1989) for an overview). *Set-based* cache partitioning can be implemented by page coloring: each partition corresponds to a disjoint subsequence of colors that maps to some disjoint subsequence of sets in the LLC. *Way-based* cache partitioning is also possible, but this requires hardware support. The ARM platform utilized in our experiments provides such support, which we describe in detail later in Sec. 3 (see Fig. 3).

Ensuring isolation with respect to memory banks. Modern DRAM designs contain multiple banks, which can be interleaved to parallelize memory accesses. Each bank consists of memory in an array of rows and columns, along with a row buffer. For a memory location to be read or written via the data bus, that location's row must be stored in the row buffer. If the row was already in the buffer, then we have a *row-buffer hit*, otherwise we have a *row-buffer miss*. In the event of a miss, the row previously in the buffer must be copied back to the array. Row-buffer misses create extra latency. Tasks executing on different processors can be prevented from causing each other to experience such misses by partitioning DRAM banks among processors (Liu et al. 2012).

Prior related work. This work follows a long line of research examining shared-resource contention in real-time systems (Kotaba et al. 2013). The use of cache partitioning in real-time systems has been considered before in the context of uniprocessor (Altmeyer et al. 2014) and multicore (Xu et al. 2016; Kim et al. 2013) platforms. However, these papers do not consider MC systems. While cache partitioning ostensibly affords strong isolation guarantees with respect to the LLC, Valsan et al. (2016) recently showed that contention may still exist on some architectures due to accesses of special cache-related registers called *miss status holding registers* (*MSHRs*). As an alternative to cache partitioning, a technique called *cache lockdown* can be used that prevents designated cached data or instructions from being evicted (Campoy et al. 2001).

Regarding memory-related issues generally, prior efforts have focused on DRAM controllers (Audsley 2013; Jalle et al. 2014; Kim et al. 2015; Krishnapillai et al. 2014), 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. 2016), accurately predicting

DRAM access delays (Kim et al. 2014), throttling lower-criticality tasks' memory accesses (Yun et al. 2012), and allocating memory (Yun et al. 2014).

Of the just-cited papers, only four consider the notion of MC scheduling espoused by Vestal (Audsley 2013; Hassan and Patel 2016; Jalle et al. 2014; Kim et al. 2015). These papers focus on hardware issues and only peripherally touch on the issue of *achieving better platform utilization from a schedulability point of view through less pessimistic analysis assumptions* as proposed by Vestal. This issue is the subject of three prior MC^2-related papers by our group, two of which (Chisholm et al. 2015; Kim et al. 2016b) are precursors to this one. The other of these papers (Ward et al. 2013) considers a scheduling-based approach to LLC management for high-criticality tasks only. In that work, the OS prefetches all potentially accessed pages before a high-criticality job executes, and enforces that co-scheduled jobs do not conflict in the LLC. Lower-criticality jobs are allowed to execute in periods of high LLC contention that otherwise would have idled a processor.

In contrast, we take a more holistic approach to hardware management here, and consider hardware-management tradeoffs within and among all criticality levels. Our research breaks new ground on several fronts. First, we are the first to investigate sharing and isolation tradeoffs with respect to shared hardware in a criticality-cognizant way (prior work only emphasized isolation). Second, we consider systems with more than two criticality levels (as opposed to only two, as in almost all prior work). Third, we are the first to investigate cache partitioning in MC systems, and in systems in which both partitioned and global schedulers are used. Fourth, we are the first to consider interference with respect to both the LLC and DRAM memory banks in MC multicore systems. Fifth, we are the first to consider the optimization problem of allocating cache space among tasks in MC systems. Finally, we are the first to address shared-hardware interference due to the OS.

3 Implementation

We now describe the hardware-management extensions[8] we added to MC^2. To discuss the specific hardware resources to be managed, we must first describe our considered hardware platform, which is a quad-core ARM Cortex A9 machine. Each core on this machine is clocked at 800MHz and has separate 32KB L1 instruction and data caches. Additionally, the LLC is a shared, unified 1MB 16-way set-associative L2 cache. 1GB of off-chip DRAM is available, and this memory is partitioned into eight 128MB banks. The basic architecture is illustrated in Fig. 2.

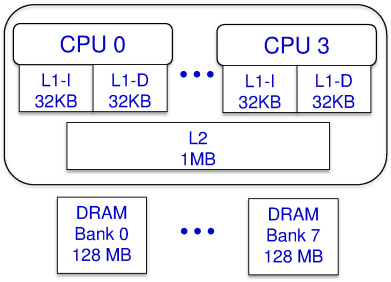
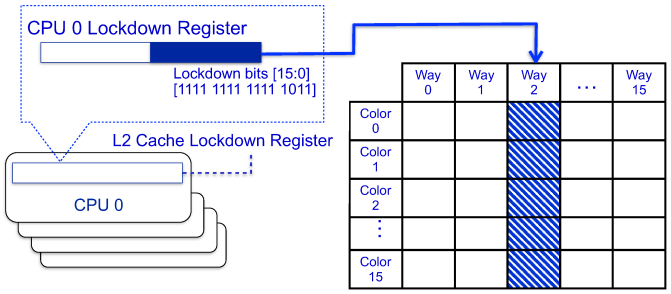
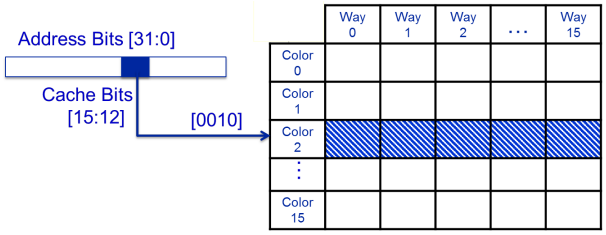


Fig. 2 Quad-core ARM Cortex A9.

[8] All source code for our new MC^2 framework is available online at `https://wiki.litmus-rt.org/litmus/Publications`.



(a) Way-based partitioning.



(b) Set-based partitioning.

Fig. 3 LLC partitioning.

Way- and set-based LLC partitioning. Our ARM platform provides per-CPU *lockdown registers* that enable the LLC to be partitioned by way. This is illustrated in Fig. 3(a). In the depicted situation, the lockdown bit corresponding to Way 2 is cleared on CPU 0, which directs cache allocations from CPU 0 to Way 2 of the LLC. The per-CPU lockdown registers are supported by an ARM Level 2 Cache Controller (L2C-310 or PL310), which is used on most Cortex A9 platforms.

As an alternative to way-based partitioning, our implementation allows set-based partitioning via page coloring. This is illustrated in Fig. 3(b) for our ARM platform, which has an LLC with 16 colors and a page size of 4KB. Way- and set-based partitioning can be combined to flexibly create rectangular LLC areas that can be designated for the sole use of certain tasks. This is illustrated in Fig. 6, which depicts our various allocation strategies; we consider this figure in detail later.

DRAM banks. Our test platform allows DRAM bank interleaving to be optionally enabled. With bank interleaving enabled, successive pages map to different banks; with it disabled, the first 32K pages map to Bank 0, the next 32K to Bank 1, and so on. Bank interleaving results in increased memory throughput in certain general-purpose applications that gain additional memory-level parallelism from interleaving. However, when enabled on our test platform, the bits within a physical address that determine the mapped-to bank overlap those that determine the LLC color, and as a result, each bank

contains pages of only two LLC colors. In contrast, with interleaving disabled, each bank contains pages of all LLC colors. The latter permits more fine-grained control over page allocations, so we disable bank interleaving. However, when allocating pages to tasks, we attempt to distribute a task's pages across all of the banks that it can access (if more than one), to obtain the benefits of bank interleaving. The manner in which we allocate pages is discussed next.

Allocating pages to tasks. A memory location's physical address determines both its LLC color and DRAM bank. To properly allocate LLC colors and DRAM banks to tasks, we construct pools of pages for each color and bank combination. We then reallocate pages to tasks from these pools via a system call. (Since MC^2 is built on LITMUS$^{\mathrm{RT}}$, an extension of Linux, reallocation is an easier way of handling page allocations for real-time tasks than modifying the memory-management subsystem in the kernel.) For a given task and assignment of colors to the task, this system call iterates through the page pools of the assigned colors in round-robin order. A page is allocated to the task from each pool in turn.

Fig. 4 shows a possible outcome of this reallocation for a task assigned colors 0–3. The system call that reallocates pages is performed before tasks commence execution, and as a result, they incur no runtime overheads due to page reallocations. In our experiments, we were able to fully allocate to the page pools all pages from four of the DRAM banks, Banks 3 through 6. Each of these four banks is dedicated to the Level-A and -B tasks on a specific CPU of our quad-core platform. The other four banks are shared by the OS and all Level-C tasks. As a result, the OS can allocate pages only from these banks and dynamic memory allocation is only supported for Level-C tasks.[9] With the exception of a rarely accessed signal-handling page, our page-coloring process can color all pages associated with each task. However, we do not allocate shared pages, though shared libraries can be dealt with via static linking. In recent work (Chisholm et al. 2016), we extended the MC^2 variant considered herein to support data sharing among tasks via shared memory, and in other recent work (Kim et al. 2016a), we further extended it to support the sharing of libraries via replicated pages; such extensions are beyond the scope of this paper.

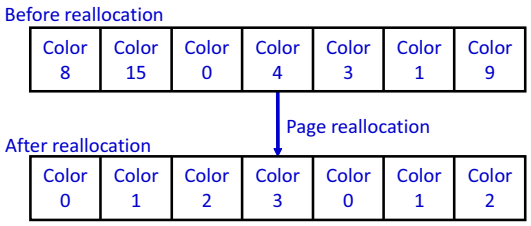


Fig. 4 Reallocation of seven pages to a task assigned the first four LLC colors. Each box represents a page.

Way-based OS isolation. Our prototype isolates the OS from Level-A and -B tasks in the LLC via way-based partitioning. Specifically, whenever the kernel begins executing on a CPU as the result of an interrupt, exception, or system call, we save the current

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

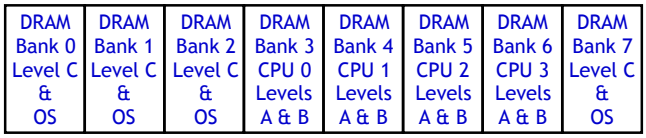


Fig. 5 DRAM allocation strategy.

value of that CPU's lockdown register and then modify it so that the OS accesses only certain LLC ways in kernel mode. When exiting kernel mode, we restore the lockdown register using the saved value. Together with the DRAM isolation just described, this ensures that the OS only minimally interferes with Levels A and B.

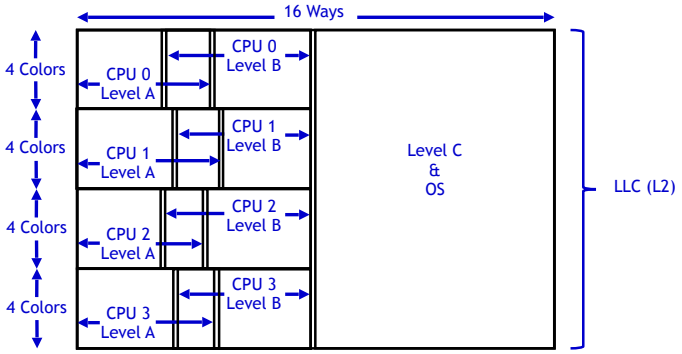
Unmanaged hardware resources. Our prototype does not provide management for L1 caches, MSHRs (Valsan et al. 2016; Kroft 1981), translation lookaside buffers (TLBs), memory controllers, or memory buses. However, under a measurement-based approach to determining PETs, such unconsidered resources are implicitly considered when PETs are determined.

Overall allocation strategy. Our DRAM allocation strategy, discussed earlier, is depicted in Fig. 5. The LLC-allocation strategies studied in this paper are depicted in Fig. 6. We categorize these allocation strategies into three separate *variants*.

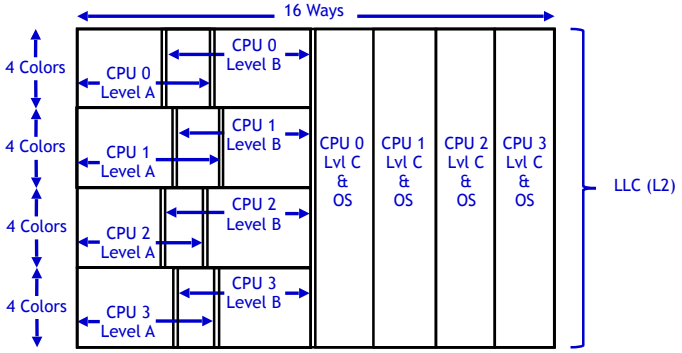
LLC-Allocation Variant 1, shown in Fig. 6(a), ensures strong isolation guarantees for higher-criticality tasks while allowing for fairly permissive hardware sharing for lower-criticality tasks when used in combination with our DRAM-allocation strategy. As seen, Level C and the OS share a subsequence of the available LLC ways and all LLC colors. As noted in Sec. 2, we assume that Level-C tasks (being SRT) are provisioned on an average-case basis. Under this assumption, LLC sharing with the OS should not be a major concern. The remaining LLC ways are partitioned among Level-A and -B tasks on a per-CPU basis. That is, the Level-A and -B tasks on a given core share a partition. Each of these partitions is allocated 1/4 of the available colors, as depicted. This scheme ensures that Level-A and -B tasks do not experience LLC interference due to tasks on other cores (*spatial* isolation). LLC interference between Level-A and Level-B tasks on the same core may be permissible, as this interference only affects Level-B tasks, as will be explained in Sec. 5.2. However, it may still be desirable to limit this interference. As a consequence, different LLC areas are allocated to Level A and B within a core partition, but these areas may overlap.[10]

Variants 2 and 3, depicted in insets (b) and (c) of Fig. 6, are analyzed in later sections to characterize the advantages and disadvantages of isolation and sharing for each criticality level. In Variant 2, the LLC area allocated to Level C and the OS is partitioned by way on a per-CPU basis. This variant provides stronger isolation guarantees to Level-C tasks but reduces the LLC area that such a task can utilize. In Variant 3, Level-A and -B tasks are not partitioned by core, thus giving each Level-A

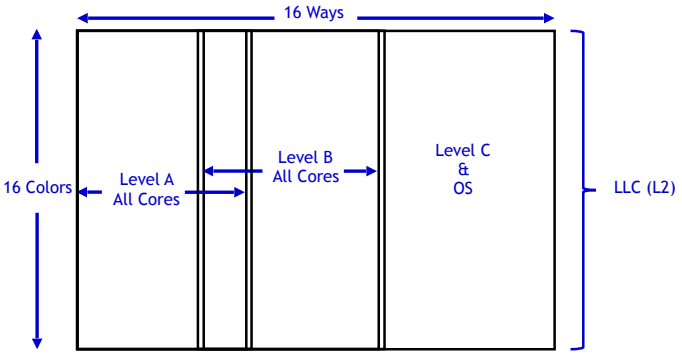
[10] We often use the term "area" instead of "partition" to describe these allocated LLC regions because of the potential for some regions to overlap.



(a) Variant 1.



(b) Variant 2.



(c) Variant 3.

Fig. 6 LLC allocation variants. LLC boundaries indicated by double lines are settable parameters. Note that the Level-A and -B LLC areas for each core can overlap in Variants 1 and 2. Overlap for all cores is allowed in Variant 3. In Variant 2, the LLC space allocated to Level C is distributed among cores as evenly as possible, since any Level-C task may run on any core. Hence, we do not tune individual per-core Level-C partition sizes independently of the overall space allocated to Level C. Because of this, we do not use double lines in indicating these per-core partition boundaries.

and -B task access to all 16 colors. This reduces isolation guarantees at Levels A and B, but increases the LLC area that Level-A and -B tasks can utilize.

The specific number of LLC ways assigned to each allocated LLC area is a tunable parameter that affects the execution times of tasks, as demonstrated later in Sec. 4. We determine values for these parameters on a per-task-set basis using optimization techniques presented in Sec. 5.3. These optimization techniques seek to minimize a task set's Level-C utilization while ensuring schedulability at all criticality levels.

4 Micro-Benchmark Experiments

We experimentally assessed the impact of combining MC allocation and hardware management in attacking the one-out-of-m problem via the following process. First, to assess the impact of hardware management, we collected extensive trace data for synthetic micro-benchmark programs and six benchmark programs. Next, to assess the impact of interference caused by the OS, we examined the impact of providing OS isolation. In this section, we present a subset of this data and resulting observations.

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

The benchmark programs we considered are listed in Table 1 and come from the *Data Intensive Systems (DIS) Stressmark Suite* (Musmanno 2003). This suite was defined to reflect memory-usage patterns common in real-world use cases.

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

Table 1 DIS Stressmark programs (Musmanno 2003).

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

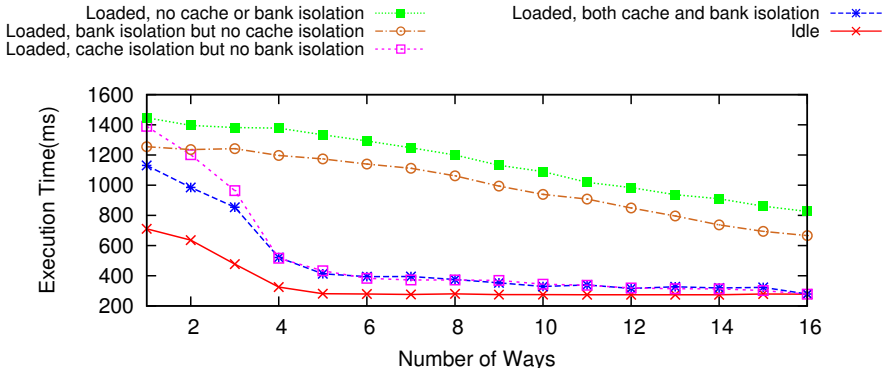
Impact of providing full isolation at Levels A and B. In our first set of experiments, we examined the impact of providing full isolation to a task by giving it a dedicated LLC area and/or DRAM bank that are accessed by no other task. When full isolation is provided, these experiments have implications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

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

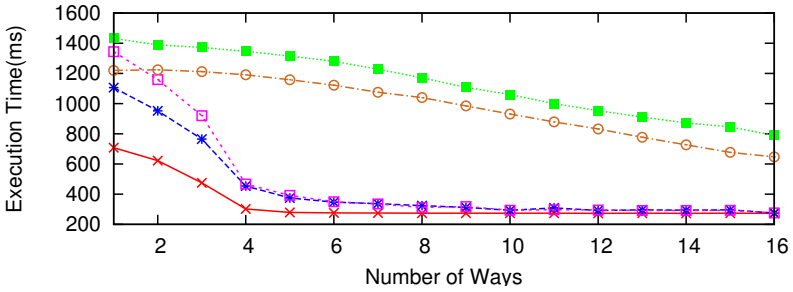
For each measured program and isolation configuration, we considered 272 possible LLC area sizes (given by 0 to 16 ways and 1 to 16 colors) for allocation to the measured program. Each additional way or color increases the allocated LLC space by 4KB. This process yielded $272 \times 5 = 1{,}360$ experiments per measured program. In each such experiment, we ran the measured program 100 times and recorded the (observed) WCET and average-case execution time (ACET) from the data collected. We are interested in both WCETs and ACETs because both are used in MC^2 provisioning, as discussed in Sec. 2.

We collected trace data (8GB in total) for all six programs in Table 1 and for micro-benchmark-programs with WSSs in $\{32, 64, 256, 512\}$KB. (Recall that the L1 caches on our hardware platform are 32KB.) We now make several observations based on this data. We support these observations using the data in Figs. 7–9, which depict recorded WCETs and ACETs for the 256KB-WSS and 32KB-WSS variants of the micro-benchmark program and the Matrix program, given an allocated LLC area consisting of 16 colors and some number of ways, and 16 ways and some number of colors. The rest of our collected data is given online (Kim et al. 2016c).

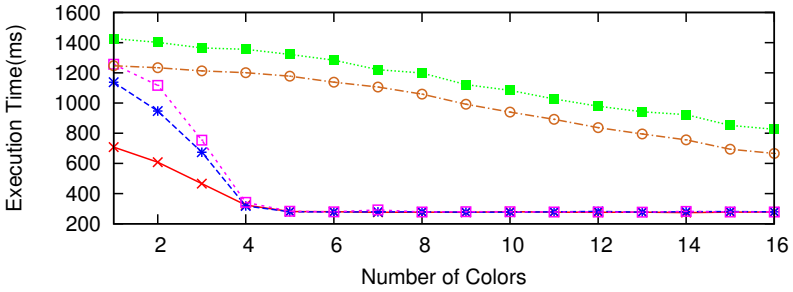
Obs. 1 *Providing LLC isolation reduced WCETs by up to* 369% *for the micro-benchmark program and by up to* 142% *for the Matrix program.*



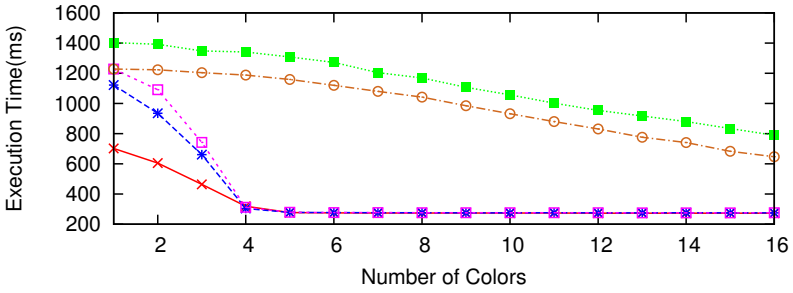
(a) WCET assuming 16 colors.



(b) ACET assuming 16 colors.

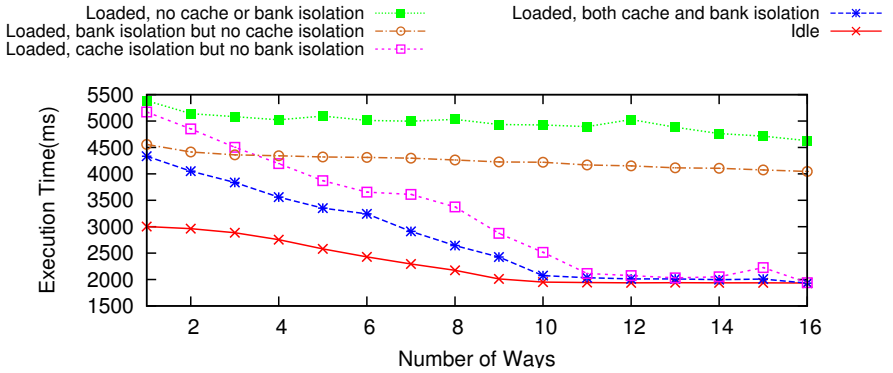


(c) WCET assuming 16 ways.

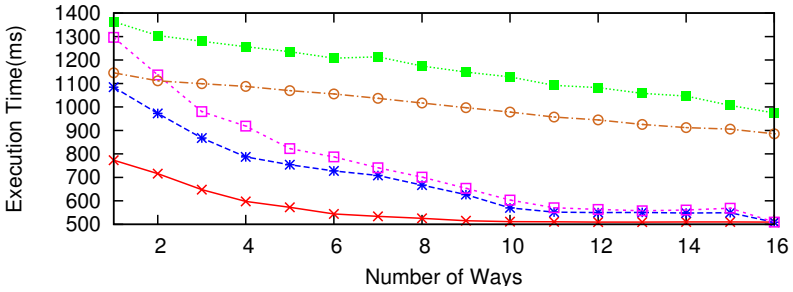


(d) ACET assuming 16 ways.

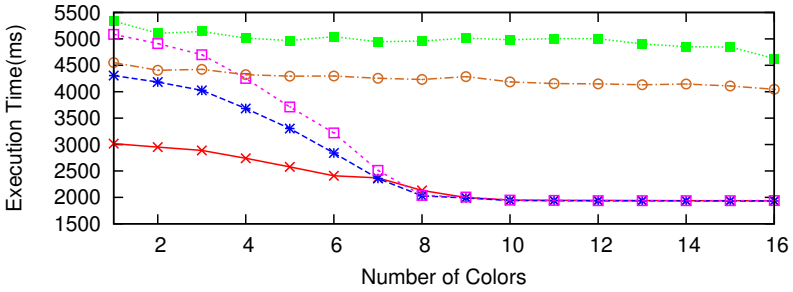
Fig. 7 Execution-time data for the 256KB-WSS micro-benchmark program.



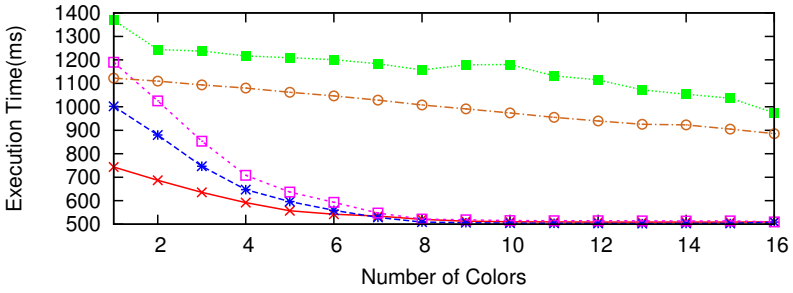
(a) WCET assuming 16 colors.



(b) ACET assuming 16 colors.



(c) WCET assuming 16 ways.



(d) ACET assuming 16 ways.

Fig. 8 Execution-time data for the Matrix program.

The noted 369% (respectively, 142%) reduction can be seen by comparing the curves in Fig. 7(c) (respectively, Fig. 8(a)) at the data points corresponding to five colors (respectively, twelve ways). However, isolation comes at a cost. For example, if we choose to isolate Level-C tasks by way on a per-CPU basis (refer to Fig. 6(b)), then each Level-C task would only be able to access 1/4 of the available LLC area size (assuming it is divisible by four) instead of sharing the entire area.

Obs. 2 *LLC isolation had a greater impact on WCETs as LLC space approached the ICAS. Beyond this point, WCETs were only marginally greater than in an idle system, implying that unmanaged hardware resources* (*TLB, MSHRs, memory bus, memory controllers—see Sec. 3*) *had only a small impact.*

In insets (a) and (c) of Fig. 7, the WCET with LLC isolation becomes quite close to that in an ideal system at four colors, which yields an LLC area matching the micro-benchmark program's WSS, and therefore its ICAS. A similar trend can be seen in Fig. 8(a) at ten ways and in Fig. 8(c) at seven colors.

Obs. 3 *Isolation with respect to both the LLC and DRAM banks improved WCETs over LLC isolation alone especially when the allocated LLC area is less than the ICAS.*

This effect can be seen in insets (a) and (c) of Fig. 7 and in insets (a) and (c) of Fig. 8. Note that, if the allocated LLC area is at least the given program's ICAS, then DRAM bank isolation has only a small impact.

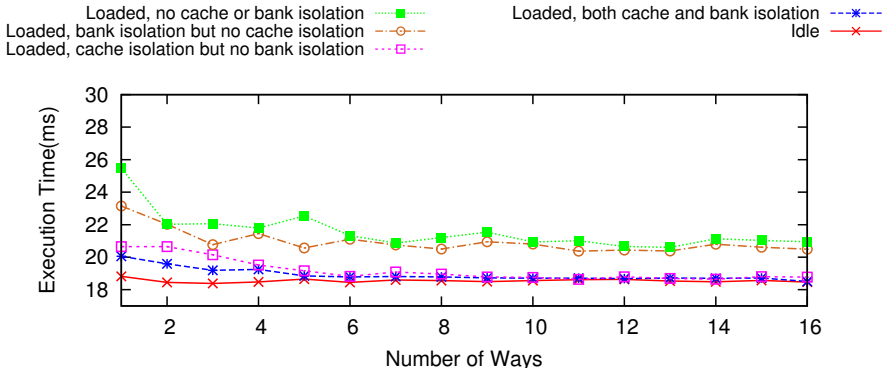
Recall from Sec. 2 that Level-A and -B tasks are included in the Level-C analysis, which assumes Level-C PETs for all tasks. Thus, ACETs are important to consider for all tasks.

Obs. 4 *The WCET trends noted in Obs. 1–3 also apply to ACETs. ACETs were lower than WCETs by approximately 5-10% (respectively, 80%) for the micro-benchmark program (respectively, Matrix program).*

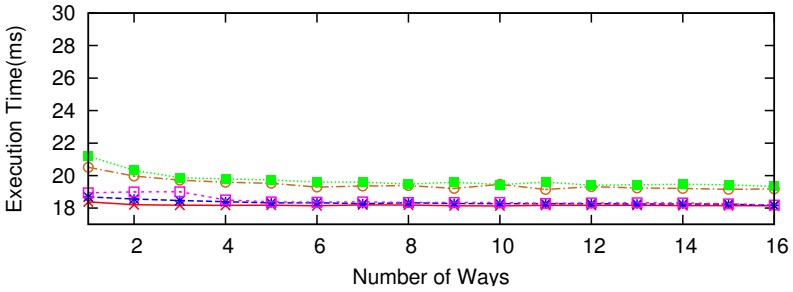
This can be seen by comparing insets (a) and (b) in Fig. 7 and insets (a) and (b) in Fig. 8. (The 5-10% reduction may be somewhat hard to see because of the scale.) The Matrix program exhibits a relatively lower ACET because it is less deterministic than the micro-benchmark program.

Obs. 5 *The execution times of the 32KB-WSS micro-benchmark program were anomalously high for some allocated LLC area sizes.*

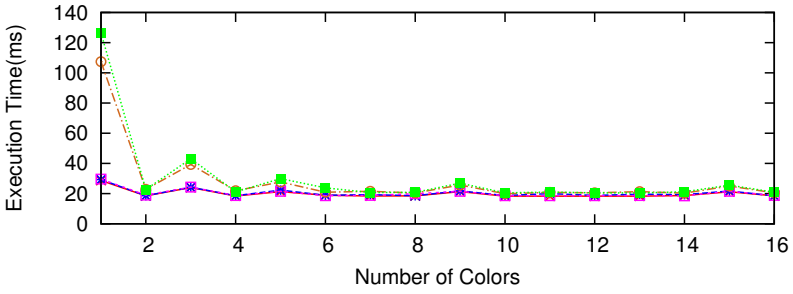
We originally expected that since this WSS matches the size of the L1 data cache, all memory references would hit in the L1 caches with the exception of compulsory misses. However, the results in insets (c) and (d) of Fig. 9 do not support this hypothesis. Observe that, in these insets, some odd-numbered color allocations resulted in "spikes" in the execution times. The baseline execution time of $20\,\mathrm{ms}$ corresponds to all references hitting in the L1 caches, as shown in insets (a) and (b) of Fig. 9, so these spikes are due to L1 cache misses. To explain these spikes, we must carefully consider the cache structure.



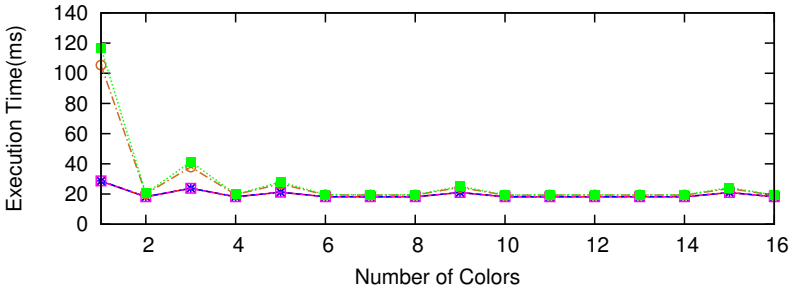
(a) WCET assuming 16 colors.



(b) ACET assuming 16 colors.



(c) WCET assuming 16 ways.



(d) ACET assuming 16 ways.

Fig. 9 Execution-time data for the 32KB-WSS micro-benchmark program.

Our L1 data cache is a 32KB 4-way set-associative cache and has two colors, which we denote here as *X* and *Y* to distinguish them from the LLC colors. For each physical address, the least significant bit of its LLC color bits determines its L1-data-cache color. Even-numbered LLC colors are mapped to Color X and odd-numbered LLC colors are mapped to Color Y. The WSS of the considered micro-benchmark program is 32KB, so there are eight data pages to be colored. The spikes were observed when the required LLC colors were not equally distributed between the L1-Colors X and Y. For example, the eight pages of the micro-benchmark program could be allocated using the first three LLC colors as follows: three pages of Color 0, three pages of Color 1, and two pages of Color 2. In this case, as shown in Fig. 10, five pages are mapped to to the L1-Color X and three pages are mapped to the L1-Color Y. Since our L1 cache is only 4-way set associative, the five pages of Color X compete for four ways, thereby causing evictions. This behavior was not observed for all odd-numbered color allocations.

Fig. 10 Conflict in the L1 cache.

(a) A case where the color starts from 3.

(b) A case where the color starts from 4.

Fig. 11 Two potential mappings of eight data pages with nine LLC colors. White regions are mapped to Color X while gray regions are mapped to Color Y.

For instance, suppose the micro-benchmark program is given an LLC area with Colors 0 through 8. Fig. 11 shows two potential mappings of the program's eight pages to LLC colors. In the first mapping, illustrated in Fig. 11(a), four pages are mapped to Color X (denoted in white) and four pages are mapped to Color Y (denoted in gray), and L1 evictions do not occur. In the second mapping, illustrated in Fig. 11(b), LLC-Color 3 is not used. As a result, there are five pages of Color X and three pages of Color Y, which again causes evictions, explaining the execution-time spikes. The presence of these execution-time spikes where none was expected provides a cautionary note for designers concerned with provisioning systems. In particular, a complete understanding of the caching and memory characteristics of the considered platform is needed in order to avoid producing execution-time estimates that are problematic. With respect to the results discussed in later sections, no special consideration of execution-time spikes was required because all of the allocation schemes we considered provide an even number of colors to all tasks.

Space tradeoffs. While the data discussed above shows clear execution-time benefits from isolation, this does not address the additional LLC space constraint imposed by

partitioning. Here, we compare specific execution times associated with the different LLC allocation variants shown in Fig. 6. For example, suppose we can allocate the entire LLC to Levels A and B under Variant 1 or 3. From the perspective of the micro-benchmark program considered in Fig. 7(c), these two allocation choices can be examined by comparing the curve for a loaded system with LLC and DRAM bank isolation at four colors to that for a loaded system with no LLC isolation, but DRAM bank isolation, at 16 colors. By comparing these two points, we see that the WCETs under these two allocations are $317\,\mathrm{ms}$ and $665\,\mathrm{ms}$, respectively, a 52% WCET improvement under isolation at Levels A and B. Similarly, suppose we can allocate the entire LLC to Level C under Variant 1 or 2. From the perspective of the micro-benchmark program considered in Fig. 7(b), these two allocation choices can be examined by comparing the curve for a loaded system with no LLC isolation at 16 ways to that for a loaded system with LLC isolation at four ways (we assume that bank isolation is not provided, as is the case for Level C). By comparing these two points, we see that the ACETs for these two options are $677\,\mathrm{ms}$ and $466\,\mathrm{ms}$, respectively, a 31% ACET improvement under isolation at Level C. To more clearly investigate the execution-time differences afforded by sharing versus isolation, we generated histograms, four of which are given in Fig. 12. The manner in which each histogram should be interpreted is explained in the figure's caption.

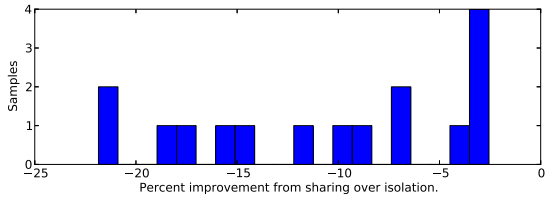
Obs. 6 *WCETs under per-core partitioning at Levels A and B were almost always lower than WCETs under cross-core sharing of the available LLC space.*

Insets (a) and (b) of Fig. 12 show that isolation improves WCETs in all cases for the Matrix and 256KB-WSS micro-benchmark programs. Similar results were found for other programs. Given that isolation is overwhelmingly preferable to sharing at Levels A and B, we chose not to consider Variant 3 of our general allocation strategy illustrated in Fig. 6(c) in the schedulability experiments presented later.

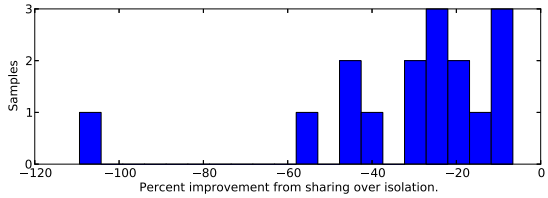
Obs. 7 *For a given LLC allocation, sharing an entire allocated LLC area at Level C without isolation and partitioning that area and providing isolation are incomparable with respect to ACETs. This exposes a tradeoff that is dependent upon the given task system to be supported, as well as the size and configuration of the LLC-allocation space under consideration.*

Fig. 12(c) shows that for the Matrix program, isolation tends to improve ACETs, while Fig. 12(d) shows that for the 256KB-WSS micro-benchmark program, the reverse is true. Also, there exist cases in which one approach may be substantially better, as demonstrated by the leftmost "outlier" in Fig. 12(d), where isolation yields a 45% improvement. Given this tradeoff, we opted to fully consider in the schedulability experiments presented later Variant 2 of our allocation strategy, shown in Fig. 6(b), in which the Level-C/OS LLC area is partitioned instead of shared.

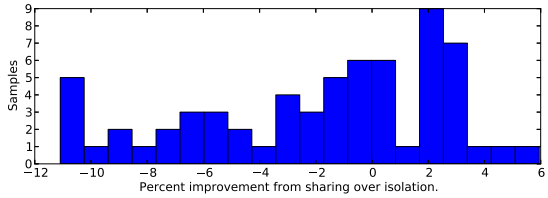
Impact of sharing at Level C: additional considerations. If hardware isolation is provided, then ACETs can be computed without much regard for the background workload. However, the situation is murkier for Level-C tasks assuming Variant 1 of the LLC-allocation strategy in Fig. 6 since Level-C tasks share the same LLC area and



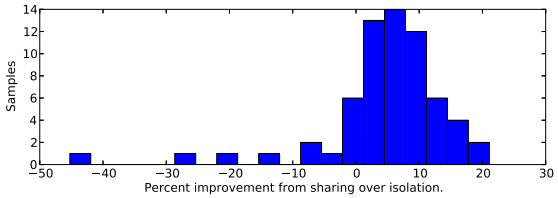
(a) Matrix WCETs.



(b) 256KB-WSS micro-benchmark WCETs.



(c) Matrix ACETs.



(d) 256KB-WSS micro-benchmark ACETs.

Fig. 12 Histograms showing the percentage improvement in the ACETs and WCETs of one micro-benchmark program and one benchmark program provided by sharing the program's allocated LLC area, instead of ensuring isolation. The x-axis gives different percentages, and for a given percentage, the histogram shows the number of cases observed across all considered LLC-allocation sizes that exhibited that percentage improvement. Negative (respectively, positive) percentages indicate that isolation (respectively, sharing) produces lower execution times. WCETs are with respect to allocation variants (see Fig. 6) at Levels A and B with bank isolation. ACETs are with respect to allocation variants at Level C without bank isolation.

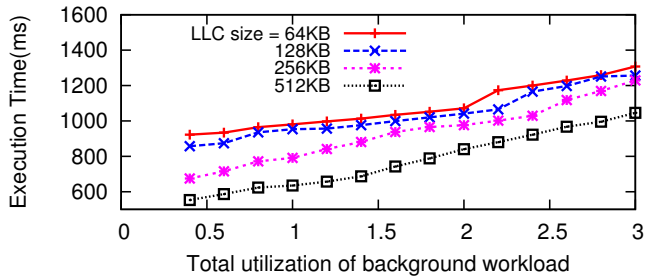


Fig. 13 ACETs of the Matrix program with varying LLC sizes and background workloads.

DRAM banks. To better understand this issue, we conducted experiments in which a mixture of DIS programs was executed at Level C and ACETs were determined for individual programs. The following observation follows from these experiments.

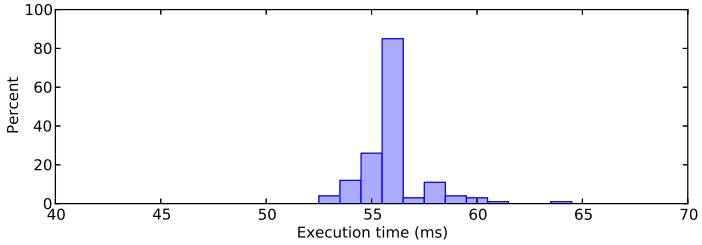
Obs. 8 *Level-C ACETs for Level-C tasks increased when the allocated Level-C LLC area was reduced or when the utilization of the background Level-C workload was increased.*

Data supporting this observation is given in Fig. 13, which gives ACETs for the Matrix program assuming various LLC area sizes and total background Level-C utilization. To determine Level-C PETs for Level-C tasks in practice, some domain knowledge would be required when defining an appropriate background workload. In our schedulability experiments, we determined such PETs by "indexing into" a graph similar to that in Fig. 13, which is reflective of the assumption that the background workload is a mix of DIS programs. In Sec. 6.2, we present the results of runtime experiments involving observed timeliness that partially validate this and other provisioning assumptions made in this paper.

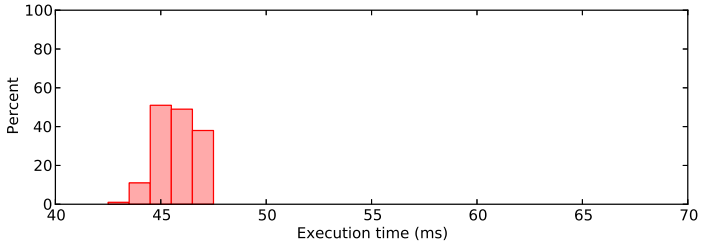
OS isolation. Our new MC^2 prototype isolates the OS from Levels A and B with respect to the LLC and DRAM banks (recall Figs. 5 and 6). As a result, execution within the OS does not cause any interference of Level-A and -B tasks with respect to the LLC or DRAM banks. We examined the impact of this feature by conducting an experiment in which a Level-B micro-benchmark program was executed with and without OS isolation. The micro-benchmark program was modified to invoke a dummy system call once per loop iteration that allocated and read 16 pages of memory. While such a system call may seem somewhat extreme, the point here is that if OS isolation is not provided, then predictability can be lost, unless the code paths the OS will take are known with high assurance. Fig. 14 shows measured execution times for 100 jobs of the micro-benchmark program, with and without OS isolation.

Obs. 9 *OS isolation reduced the WCET and ACET of the micro-benchmark program considerably.*

The WCET (respectively, ACET) decreased by 24% (respectively, 16%) in Fig. 14. The execution of the dummy system call can evict cache lines of 16 pages of the



(a) OS not isolated.



(b) OS isolated.

Fig. 14 A histogram of execution times for 100 jobs of a Level-B micro-benchmark program with and without OS isolation.

micro-benchmark program, which results in cache misses when user-mode execution resumes. OS isolation prevents evictions of user pages at Levels A and B in kernel mode since the OS only accesses the Level-C/OS partition.

In the experiment considered above, OS code executes on the same core as the program invoking the system call. In addition to such same-core interference, the OS can also cause cross-core interference. In particular, because OS pages are spread over all cache colors, they can cause LLC evictions of tasks on any core if not managed. To demonstrate the potential ill effects of cross-core OS interference, we executed two micro-benchmark programs concurrently at Level B on different cores. One program, referred to as the *caller*, issued dummy system calls after each loop iteration, while the other program, referred to as the *victim*, did not. The caller ran on Core 0 while the victim ran on Core 1. The two programs were allocated separate LLC partitions and DRAM banks. Recorded WCETs and ACETs for these programs, with and without OS isolation, are shown in Fig. 15.

Obs. 10 *OS isolation reduced the WCET and ACET of both the caller and victim by 20%.*

These results suggest that isolating user-space programs from each other with respect to the LLC and DRAM banks alone is not sufficient to mitigate execution-time interference; the OS must be isolated as well.

OS-related overheads can induce pessimism in schedulability analysis (Brandenburg 2011), but Figs. 14 and 15 suggest that per-overhead costs can be significantly

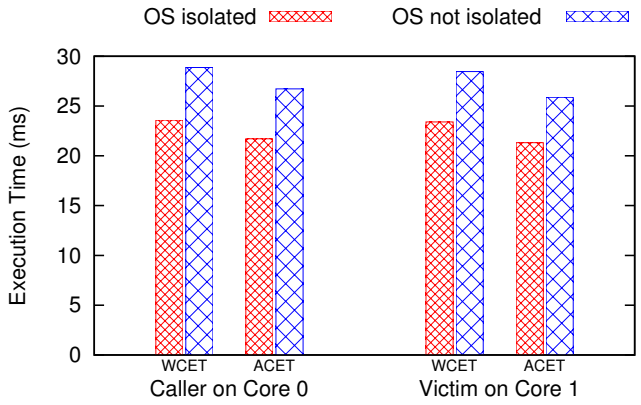


Fig. 15 Cross-core OS interference.

reduced through OS isolation. One potential concern, however, is that restricting the OS to execute within a smaller LLC area might increase its own execution times unacceptably. However, in additional experiments, we found that providing OS isolation increased system-call overheads by a mere $35\,\mathrm{ns}$ in the worst case and by $15\,\mathrm{ns}$ on average. Such small overheads are negligible in comparison with the noted reductions in WCETs and ACETs.

While our OS-isolation technique prevents the OS from evicting data from Level-A or -B tasks, it does not address communication between a Level-A or -B task and the kernel itself. Consider, for example, the `read()` system call, in which the kernel copies data from kernel space into user space. When the kernel copies data into user-space pages, it will load the data into OS LLC ways, which are subject to interference from Level-C tasks and other kernel invocations. In future work, we intend to fully consider this issue and other issues that arise when Level-A and -B tasks use OS services.

In this section, we have provided recommendations for how to handle several tradeoffs (such as tradeoffs for cross-core isolation vs. sharing in the LLC at Levels A and B), but other tradeoffs still require further analysis. For example, as seen in Fig. 7, providing more LLC space to Level-A or -B tasks may reduce their execution times dramatically, but this may limit the available LLC space for Level C, which could increase Level-C execution times. How do we handle this tradeoff? In Sec. 5, we show that this tradeoff can be addressed by means of an optimization framework based on linear programming. Additionally, we still must provide evidence in support of the major thesis of this paper, namely, that combining MC analysis and hardware management is more effective in attacking the one-out-of-m problem than applying either technique alone. We provide such evidence in the form of a large-scale overhead-aware schedulability study in which the aforementioned optimization framework was applied. The results of this study are discussed in Sec. 6.

5 An Optimization Framework for Determining LLC Allocations

As indicated by the double lines in Fig. 6, our proposed variants for allocating LLC space allow some flexibility. In this section, we present an optimization framework for sizing LLC areas based on linear programming. We focus particularly on Variant 1 of our allocation strategy, but our techniques can be applied to the other variants with minor modifications.

5.1 Overview of Notation and MC^2 Schedulability

In order to formulate the linear program (LP) that underlies our optimization framework, we must first refine the notation presented earlier in Sec. 2 to allow certain details to be considered that were heretofore unimportant. We consider a set of implicit-deadline periodic tasks $\tau = \{\tau_1, \tau_2, \tau_3, ..., \tau_n\}$ to be scheduled under the MC^2 framework on m cores. We only consider Levels A–C, as stated earlier. Each task τ_i has a period T_i, and three PETs, e_i^A, e_i^B, and e_i^C, where e_i^ℓ denotes its Level-ℓ PET (recall the discussion concerning MC schedulability analysis in Sec. 2). We let τ^A, τ^B, and τ^C denote the subset of tasks in τ at Levels A, B, and C, respectively. Also, we let $\tau^{A,p}$ and $\tau^{B,p}$ denote the subset of tasks in τ^A and τ^B, respectively, that are assigned to core p. We denote the total utilization of all Level-ℓ tasks assuming Level-ℓ' execution times as $U_\ell^{\ell'} = \sum_{\tau_i \in \tau^\ell} \frac{e_i^{\ell'}}{T_i}$. We denote the total utilization of all Level-A or -B tasks assigned to core p assuming Level-ℓ' execution times as $U_{A,p}^{\ell'} = \sum_{\tau_i \in \tau^{A,p}} \frac{e_i^{\ell'}}{T_i}$ and $U_{B,p}^{\ell'} = \sum_{\tau_i \in \tau^{B,p}} \frac{e_i^{\ell'}}{T_i}$, respectively. The schedulability condition for Level C is dependent on the largest Level-C utilization of any Level-C task, which we denote as h, and the sum of the $m-1$ largest Level-C utilizations among Level-C tasks, which we denote as H. The following are sufficient conditions for ensuring schedulability at all three criticality levels (Mollison et al. 2010).

$$\forall p :: U_{A,p}^A \leq 1 \wedge \tag{1}$$

$$\forall p :: U_{A,p}^B + U_{B,p}^B \leq 1 \wedge \tag{2}$$

$$U_A^C + U_B^C + U_C^C \leq m \wedge \tag{3}$$

$$U_A^C + U_B^C + (m-1)h + H < m \tag{4}$$

As noted in Sec. 2, we assume that PETs are determined via a measurement process. As shown in Sec. 4, these measurement-based PETs will generally depend on allocated LLC areas. We denote the Level-ℓ PET of task τ_i when its allocated LLC area consists of W ways and S colors (refer to Fig. 6) as $e_i^\ell(W, S)$. (We use "S" in denoting colors because colors determine LLC *sets*, and the term "C" has a predefined meaning in the context of MC^2.)

Canonical LLC allocation and problem to be solved. We consider a canonical LLC allocation, given by the LLC-allocation variant illustrated in Fig. 6(a), with respect to our quad-core ARM platform. Assuming an LLC with W^{max} ways in total, all Level-C tasks together are allocated an LLC area that consists of all colors (sets) associated with the last W_C ways, Ways $W^{max} - 1 - W_C$ through $W^{max} - 1$, for some W_C. All LLC areas for Level-A and -B tasks are taken from the colors (sets) associated with Ways 0 through $W_C - 1$. The Level-A and -B tasks on each core use an LLC area consisting of $1/m$ ($m = 4$ on our platform) of the colors (sets) associated with these ways, as depicted in Fig. 6(a). Each per-core Level-A and -B LLC area is subdivided into potentially overlapping Level-A and -B areas.

The technical LLC allocation problem considered here is to determine how to precisely size these LLC areas so as to enhance schedulability given the characteristics of the task system in question. That is, we seek to determine how the doubled lines in Fig. 6(a) should be set given execution time data, such as that in Fig. 8, for all tasks in question. In addressing this problem, we assume that an assignment of all Level-A and -B tasks to cores has already been determined.

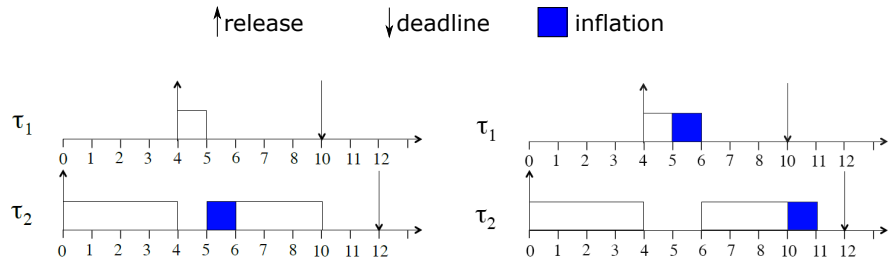
5.2 MC^2 LLC-Managed Overhead Accounting

As a precursor to setting up our LP, we begin by discussing how to alter the schedulability conditions given in (1)–(4) to account for overheads. This is important because the manner in which LLC areas are sized can affect overheads, and overheads in turn impact schedulability.

Many overhead sources can be modeled independently of LLC allocations with prior overhead-modeling techniques. Further discussion of such overhead sources can be found in Appendix A. Here, we consider a particularly important overhead that *is* dependent on LLC allocations, *cache-related preemption delays* (*CRPDs*), and how this overhead is affected by our LLC allocation methods. CRPDs are delays a task may incur to reload lines evicted from the LLC (and other caches) due to a preemption and write back to memory dirty data in these evicted lines. We discuss how to quantify CRPDs with respect to LLC allocation sizes, so that these delays can be integrated into schedulability analysis.

There are two basic ways to account for CRPD costs, as shown in Fig. 16. Under *task-centric accounting*, the execution time of the *preempted* job is inflated to account for the preemption. Under *preemption-centric accounting*, the execution time of the *preempting* job is inflated to "pay" for the CPRD cost of any preempted job that *resumes* execution when the preempting job completes. We consider preemption-centric accounting here (with one exception, discussed later), because it usually introduces less pessimism in schedulability analysis, and because it can be linearly modeled by simply adding an inflation term to each execution cost. (Task-centric accounting often entails the introduction of non-linear ceiling and/or floor operators.)

CRPD-overhead inflations. The inflation term we add to a task's execution times is generally a function of that task's allocated LLC area size. For example, we can inflate



(a) Task-centric accounting. **(b)** Preemption-centric accounting.

Fig. 16 Forms of overhead accounting.

the Level-ℓ execution time of any Level-B or -C task that has an LLC area consisting of W ways and S colors as follows:

$$\forall i : \tau_i \in \tau^B \cup \tau^C :: e'^{\ell}_i(W,S) = e^{\ell}_i(W,S) + E^{\ell}(W,S), \tag{5}$$

where $E^{\ell}(W,S)$ is the time required, according to Level-ℓ analysis, to reload all cache lines and write back any dirty lines to memory within a region of the LLC consisting of W ways and S colors. Note that this is the LLC area of *both* the preempting and preempted task: for preemptions of Level-B tasks by Level-B tasks (or Level-C tasks by Level-C tasks), the preempting job shares the same LLC area as the preempted job. We denote b^{ℓ} as an upper-bound on the worst-case time required under Level-ℓ analysis assumptions to load the lines within an LLC area consisting of only one way and one color. Under this assumption, our inflation term is

$$E^{\ell}(W,S) = W \cdot S \cdot b^{\ell}. \tag{6}$$

We now explain how to introduce inflations into the schedulability conditions (1)–(4) discussed earlier. To do so, we can substitute for each utilization term a corresponding inflated utilization term. We denote the inflated Level-ℓ utilizations of each task τ_i as $u'^{\ell}_i = \frac{e'^{\ell}_i}{T_i}$. We can then define inflated Level-B and -C utilizations as follows.

$$\forall p :: U'^{B}_{B,p} = \sum_{\tau_i \in \tau^{B,p}} u'^{B}_i$$

$$\forall p :: U'^{C}_{B,p} = \sum_{\tau_i \in \tau^{B,p}} u'^{C}_i$$

$$U'^{C}_{C} = \sum_{\tau_i \in \tau^{C}} u'^{C}_i$$

We also replace h and H in condition (4) with inflated terms h' and H'. h' is the highest inflated Level-C utilization of any Level-C task, and H' is the sum of the $m-1$ largest inflated Level-C utilizations among Level-C tasks.

Note that we do not apply the inflation described in Equation (5) to Level-A tasks. Recall from our discussion in Sec. 2 that each Level-A job (or job slice) runs non-preemptively. Under some system implementations, the execution time of each job

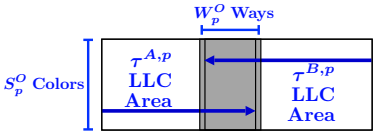


Fig. 17 Overlap for Level-A and -B LLC areas on core p.

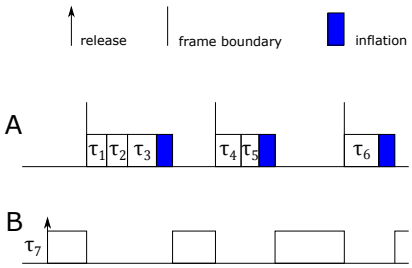


Fig. 18 Per-frame Level-B inflation for Level A.

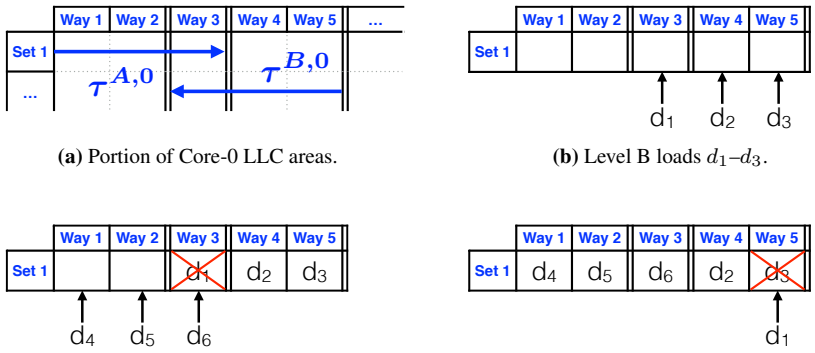
slice can be measured independently of other slices when PETs are initially determined. This ensures the PETs of one slice are not affected by cache lines loaded by other job slices. In former presentations of our work on LLC allocation optimization, we took advantage of this possibility and applied no inflation at Level A (Chisholm et al. 2015). However, under the budgeted job executions of our implementation, we do not *a priori* know the slicing point. Therefore, we must account for cache-related delays due to job slicing. The frame size for the cyclic executive of Level-A tasks on core p is equal to the smallest period of any task in $\tau^{A,p}$, which we denote $T_{A,p}^{min}$. Given this frame size, an upper bound on the number of slices per job that a task may incur is $\mathbb{s}_i = \frac{T_i}{T_{A,p}^{min}}$. We use this upper bound to define the following per-task inflated execution times and per-core utilizations:

$$\forall i : \tau_i \in \tau^A :: e'^{\ell}_i(W,S) = e^{\ell}_i(W,S) + (\mathbb{s}_i - 1) * E^{\ell}(W,S),$$

$$\forall p :: U'^{\ell}_{A,p} = \sum_{\tau_i \in \tau^{A,p}} u'^{\ell}_i.$$

This is the one case of task-centric overhead accounting in our model, since each job pays for its own CRPDs from job-slicing.

Level-A jobs may produce other CRPD overheads as well in the event that the LLC areas for Levels A and B on core p overlap. In Fig. 17, consider the shaded region representing this overlap. Level-A tasks may evict all cache lines of Level-B tasks within this region of W_p^O ways and S_p^O colors. This might suggest that the Level-B execution time of each Level-A job requires inflation to account for possible evictions in the LLC space shared with Level B. However, the required inflation can be less pessimistically determined. As shown in Fig. 18, Level-A jobs allocated to a frame run sequentially at the beginning of the frame. In this scenario, Level B is only preempted by Level A at most once per frame. Hence, an inflation is only required once per frame, as shown, rather than once per job. While some CRPDs incurred by Level B are applied to Level A, we emphasize that this preemption-centric modeling does not negatively impact the schedulability of the higher criticality Level-A tasks. These CRPD penalties are only considered when analyzing the schedulability of Level B and Level C.



(a) Portion of Core-0 LLC areas.

(b) Level B loads d_1–d_3.

(c) Level A loads d_4–d_6.

(d) Level B reloads d_1.

Fig. 19 Example of an eviction indirectly caused by Level A in a Level-B cache line isolated from Level A. Inset (a) shows the LLC areas of $\tau^{A,0}$ and $\tau^{B,0}$. First, Level B loads data d_1, d_2, and d_3 into its ways in Set 1 (inset (b)). Then, Level A evicts data d_1 in the overlap between the LLC areas of $\tau^{A,0}$ and $\tau^{B,0}$ (inset (c)). Then, Level B evicts d_3 in order to reload d_1 (inset (d)).

Depending on the replacement policy of the cache, evictions of Level-B cache lines by Level-A tasks may cause Level-B tasks to evict additional lines throughout the ways allocated to Level B. Fig. 19 shows one example of such a Level-B eviction occurring under one possible LLC configuration for $\tau^{A,0}$ and $\tau^{B,0}$. For Level B, we make the pessimistic assumption that the number of evictions directly or indirectly caused by a Level-A task is equal to the area allocated to Level B in sets it shares with Level A. For Level C, we make the more optimistic assumption that, on average, the number of evicted cache blocks is equal to the size of the overlap.

If Level B is allocated $W_{B,p}$ ways on core p, then we can model the overhead associated with reloading cache lines allocated to Level B once per Level-A frame by applying the inflation term $I^B_{A,p} = \frac{E^B(W_{B,p},S^O_p)}{T^{min}_{A,p}}$ to the Level-B utilization of $\tau^{A,p}$ when $W^O_p > 0$ and $S^O_p > 0$. If either W^O_p or S^O_p is 0, then $\tau^{A,p}$ and $\tau^{B,p}$ do not share cache lines, and thus $I^B_{A,p} = 0$. We similarly apply the inflation $I^C_{A,p} = \frac{E^C(W^O_p,S^O_p)}{T^{min}_{A,p}}$ to the Level-C utilization of $\tau^{A,p}$.

$$\forall p :: {U''}^B_{A,p} = {U'}^B_{A,p} + I^B_{A,p}$$
$$\forall p :: {U''}^C_{A,p} = {U'}^C_{A,p} + I^C_{A,p}$$

Our schedulability conditions with CRPD overheads accounted for are the following.

$$\forall p :: {U'}^A_{A,p} \leq 1 \tag{7}$$

$$\forall p :: {U''}^B_{A,p} + {U'}^B_{B,p} \leq 1 \tag{8}$$

$$\sum_{p=1}^{m} \left({U''}^{C}_{A,p} + {U'}^{C}_{B,p} \right) + {U'}^{C}_{C} \leq m \tag{9}$$

$$\sum_{p=1}^{m} \left({U''}^{C}_{A,p} + {U'}^{C}_{B,p} \right) + (m-1)h' + H' < m \tag{10}$$

5.3 Linear Program

In this section, we show how to solve the canonical LLC allocation problem described in Sec. 5.1 via an LP. The LP we obtain determines a choice of ways for each allocated LLC area such that the schedulability conditions (7)–(10) are maintained. This requires treating ways as continuous variables. We explain later how to ultimately obtain an integral solution. We now describe the various sets of constraints in our final LP.

LLC size constraints. The simplest constraint set ensures that there is no overlap between Level-C's partition and any allocated LLC areas at higher criticality levels. We let $\hat{W}_{A,p}$ and $\hat{W}_{B,p}$ denote LP variables indicating the number of ways allocated to Levels A and B, respectively, on core p. We let $\hat{W}_C$ denote an LP variable indicating the number of ways allocated to Level C. Depending on whether we are formulating a mixed-integer or non-integer linear program, we can treat these LP variables as either continuous or integer. Later in this section, we discuss in more detail the tradeoffs between the continuous and integer modeling of ways.

LLC size constraints also determine the overlap between Levels-A and -B LLC areas. $\hat{W}^{O}_{p}$ denotes a continuous LP variable modeling the overlap on core p. Recall that W^{max} is the total number of ways in the considered LLC cache. If Level-A and -B LLC areas overlap on core p, the overlap is

$$W^{O}_{p} = W_{A,p} + W_{B,p} + W_C - W^{max}.$$

Constraint Set 1. *The LLC size constraints are as follows.*

$$\forall p :: \hat{W}_{A,p} + \hat{W}_C \leq W^{max}$$

$$\forall p :: \hat{W}_{B,p} + \hat{W}_C \leq W^{max}$$

$$\hat{W}^{O}_{p} \geq \hat{W}_{A,p} + \hat{W}_{B,p} + \hat{W}_C - W^{max}.$$

Modeling execution times. In order to determine LLC allocations for which a task system is schedulable, we need to model the impact of LLC area sizes on the utilizations used in (7)–(10). A model for utilizations, in turn, requires a model for task PETs. The manner in which we model the impact of allocated LLC area sizes on PETs affects the choice of optimization techniques that can be applied to determine such sizes. In particular, our LP requires strictly linear relationships between LLC area sizes and PETs.

Execution-time measurements, such as those required for PETs under MC^2, often exhibit a property that we will exploit:

Execution-Time Assumption. *The derivative of a task's execution time* (*at any level*) *with respect to its allocated LLC area size is non-increasing. That is, the execution-time function is non-convex.*

Bui et al. (2008) presented graphs for execution times of several avionics applications that approximately meet this condition, suggesting that this behavior is not uncommon. The measurement data discussed in Sec. 4 exhibits similar behavior. We note three properties that directly follow from this assumption.

Lemma 1 *The derivative of a task's* ***inflated*** *execution time* (*at any level*) *with respect to its allocated LLC area size is non-increasing.*

Proof For our LLC allocation problem, colors are fixed at each level, such that the execution-time function for each task τ_i is a function of the number of ways allocated to τ_i. By (6), the inflation function E^ℓ varies linearly with allocated ways, and is thus non-convex. The sum of two non-convex functions is non-convex.□

Lemma 2 *The derivative of a task's inflated* ***utilization*** (*at any level*) *with respect to its allocated LLC area size is non-increasing.*

Proof This follows from the fact that task utilizations are directly proportional to task execution times.□

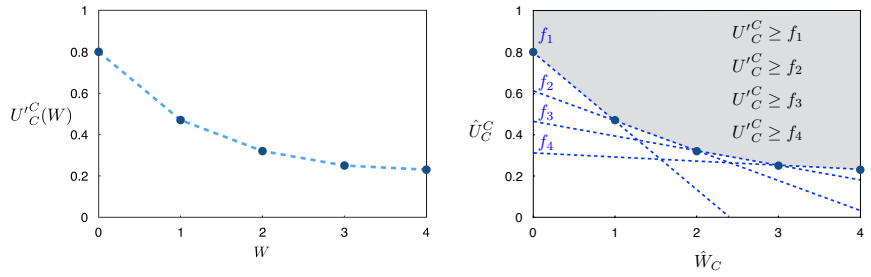
We could proceed with the construction of our LP by treating individual task utilizations as variables, but this would entail having $O(n)$ variables. We can limit the number of variables to $O(m)$ by instead considering the combined utilizations of sets of tasks. This is supported by one final property.

Lemma 3 *The derivative of the inflated utilizations (at any level) of a* ***set*** *of tasks with respect to their allocated LLC area size is non-increasing.*

Proof As stated earlier, the sum of non-convex functions is non-convex.□

While some of the assumptions made here concerning execution times may result in over-approximations of such execution times so that these assumptions are met, we have determined via schedulability studies in prior work that our LLC allocation methods still yield substantial schedulability improvements (Chisholm et al. 2015).

PET- and overhead-based constraints. Consider the hypothetical utilization plot shown in Fig. 20(a) for ${U'}_C^C$ with respect to some integer number of allocated LLC ways W. We can construct such a plot from execution-time measurement data (like that presented in Sec. 4), known task periods, and known values for b^B and b^C. In Fig. 20(b), we create a set of lines from each pair of adjacent data points, using the standard two-point line formula $f(x) = f(x_0) + (x - x_0)(f(x_0 + 1) - f(x_0))$. This is the formula for the line that contains the points $(x_0 + 1, f(x_0 + 1))$ and $(x_0, f(x_0))$. We can describe the value of f over a continuous domain with LP variables $\hat{f}$ and $\hat{x}$ constrained by such lines. Let x^{max} denote the maximum value of x for which we



(a) Utilization. **(b)** LP model of utilization.

Fig. 20 Determining LP constraints from utilizations. Inset (a) shows an example of the Level-C utilization for a task set with respect to the number of ways allocated at Level C. Inset (b) shows LP-constraint functions derived from the utilization values in inset (a). The shaded region is the continuous region in which $\hat{U}^C_C$ and $\hat{W}_C$ are constrained in the LP.

have a data point for $f(x)$. A value can be determined for $\hat{f}$ by solving the following LP.

$$
\begin{aligned}
&\text{minimize } \hat{f} \\
&\text{subject to:} \\
&\forall x \in \{0, 1, ..., x^{max} - 1\} : \\
&\hat{f} \geq f(x) + (\hat{x} - x)(f(x+1) - f(x)) \\
&\hat{f} \geq 0 \\
&0 \leq \hat{x} \leq x^{max}
\end{aligned}
$$

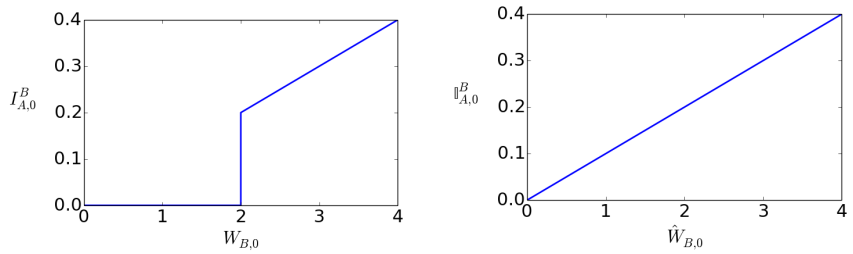
If this LP produces an integer value for $\hat{x}$, then $\hat{f}$ will equal $f(\hat{x})$. In the case considered in Fig. 20(b), our discrete function is $U'^C_C(W)$, for which we define the LP variable $\hat{U}^C_C$ to describe $U'^C_C(W)$ over a continuous domain.

$$\hat{U}^C_C \geq U'^C_C(W) + (\hat{W}_C - W)(U'^C_C(W+1) - U'^C_C(W))$$

Note that $\hat{W}_C$ is the only variable in the right-hand-side expression above, *i.e.*, this is a linear expression. We define similar LP variables $\hat{U}^A_{A,p}$, $\hat{U}^B_{A,p}$, $\hat{U}^C_{A,p}$, $\hat{U}^B_{B,p}$, and $\hat{U}^C_{B,p}$ for the inflated utilizations of Levels A and B for each core p.

The LP constraints for $\hat{U}^B_{A,p}$ and $\hat{U}^C_{A,p}$ must also include the impact of per-core inflations $I^B_{A,p}$ and $I^C_{A,p}$. Consider the following constraints for $\hat{U}^B_{A,p}$ and $\hat{U}^C_{A,p}$:

$$
\begin{aligned}
\hat{U}^B_{A,p} &\geq U'^B_{A,p}(W) + (\hat{W}_{A,p} - W)(U'^B_{A,p}(W+1) - U'^B_{A,p}(W)) + \mathbb{I}^B_{A,p} \\
\hat{U}^C_{A,p} &\geq U'^C_{A,p}(W) + (\hat{W}_{A,p} - W)(U'^C_{A,p}(W+1) - U'^C_{A,p}(W)) + \mathbb{I}^C_{A,p}
\end{aligned}
$$



(a) Overhead value. **(b)** LP model.

Fig. 21 (a) Value of $I^B_{A,0}$ with respect to $W_{B,0}$, assuming $W_C + W_{A,0} = m - 2$, $S^O_0 = 4$, $T^{min}_{A,0} = 1$ ms, and $b^B = 0.025$ ms. Under this scenario, LLC areas allocated to $\tau^{A,0}$ and $\tau^{B,0}$ overlap when $W_{B,p} > 2$. **(b)** Linear model of $I^B_{A,0}$ in the LP. An inflation is applied for all values of $\hat{W}_{B,p}$ in the LP model to avoid the non-linear "jump" at 2 ways seen in inset (a).

$\mathbb{I}^B_{A,p}$ and $\mathbb{I}^C_{A,p}$ denote LP terms for $I^B_{A,p}$ and $I^C_{A,p}$, respectively. These LP terms equate to the following LP expressions, based on our definitions of $I^B_{A,p}$ and $I^C_{A,p}$:

$$\mathbb{I}^B_{A,p} = \frac{E^B(\hat{W}_{B,p}, S^O_p)}{T^{min}_{A,p}}$$
$$\mathbb{I}^C_{A,p} = \frac{E^C(\hat{W}^O_p, S^O_p)}{T^{min}_{A,p}}$$

Note that, for an LLC with S^{max} colors, $S^O_p = S^{max}/m$ on each core p in the LLC-allocation variant depicted in Fig. 6(a). At Level B, an inflation is applied even without overlap. This conservatively models Level-A inflations at Level B to avoid non-linear constraints (see Fig. 21). This completes the LP variable relations needed to describe constraints derived from measured execution times.

Constraint Set 2. *The linear constraints for utilization variables based on task execution-time data with CRPD overheads added are as follows.*

$$\begin{aligned}
&\forall W \in \{0, 1, ..., W^{max} - 1\} :: \\
&\quad \hat{U}^C_C \geq \mathbb{U}^C_C(\hat{W}_C, W) \\
&\quad \forall p \in \{1, ..., m\} :: \\
&\qquad \hat{U}^B_{B,p} \geq {U'}^A_{B,p}(W) + (\hat{W}_{B,p} - W)({U'}^A_{B,p}(W+1) - {U'}^A_{B,p}(W)) \\
&\qquad \hat{U}^C_{B,p} \geq {U'}^B_{B,p}(W) + (\hat{W}_{B,p} - W)({U'}^B_{B,p}(W+1) - {U'}^B_{B,p}(W)) \\
&\qquad \hat{U}^A_{A,p} \geq {U'}^A_{A,p}(W) + (\hat{W}_{A,p} - W)({U'}^A_{A,p}(W+1) - {U'}^A_{A,p}(W)) \\
&\qquad \hat{U}^B_{A,p} \geq {U'}^B_{A,p}(W) + (\hat{W}_{A,p} - W)({U'}^B_{A,p}(W+1) - {U'}^B_{A,p}(W)) + \mathbb{I}^B_{A,p} \\
&\qquad \hat{U}^C_{A,p} \geq {U'}^C_{A,p}(W) + (\hat{W}_{A,p} - W)({U'}^C_{A,p}(W+1) - {U'}^C_{A,p}(W)) + \mathbb{I}^C_{A,p}
\end{aligned}$$

Modeling h and H. To construct an LP that applies all schedulability conditions to task systems, linear constraints are also required for quantities specific to Expression (10). We let $h'(W)$ and $H'(W)$ denote the values of h' and H', respectively, when W ways are allocated to Level C. We let $\hat{h}$ and $\hat{H}$ be our LP variables for $h'(W)$ and $H'(W)$, respectively. Our constraints for these variables are constructed in a similar fashion to the constraints for utilization variables. Values for $h'(W)$ and $H'(W)$ are determined from measured execution times for each integer number of ways W allocated to Level C after inflation. Linear constraints are then constructed from adjacent data points.

This requires $h'(W)$ and $H'(W)$ to be non-convex as well. These data functions, in fact, *are* non-convex under our Execution Time Assumption. To see this, let $\tau_h(W)$ denote the Level-C task with the highest inflated utilization when W ways are allocated to Level C. Note that, for any two way values, W_1 and W_2, $\tau_h(W_1)$ may indicate a different task from $\tau_h(W_2)$, since Level-C task utilizations may vary with the number of ways allocated. If $\tau_h(W)$ happens to denote the same task for different values of W, then the non-convexity of $h'(W)$ follows trivially, because the Level-C utilization of any task is non-convex. The more interesting possibility is that the task denoted by $\tau_h(W)$ changes with W. In this case, consider way values W and $W+1$ such that $\tau_h(W) \neq \tau_h(W+1)$. This implies that the derivative of $\tau_h(W+1)$'s utilization is greater than the derivative of $\tau_h(W)$'s utilization at W. Hence, the derivative of h' is greater at $W+1$ than W, and h remains non-convex at $W+1$.

By similar logic, $H'(W)$ is guaranteed to be non-convex.

Constraint Set 3. *The linear constraints for $\hat{h}$ and $\hat{H}$ based on measured task-set utilizations with CRPD overheads are as follows.*

$$\begin{aligned}
&\forall W \in \{0, 1, ..., W^{max} - 1\} :: \\
&\quad \hat{H} \geq h'(W) + (\hat{W}_C - W)(h'(W+1) - h'(W)) \\
&\quad \hat{h} \geq H'(W) + (\hat{W}_C - W)(H'(W+1) - H'(W))
\end{aligned}$$

Schedulability constraints. To fully characterize all constraints on utilizations and ways, we must include the schedulability constraints based on Expressions (7)–(10). Expression (10) is a strict inequality. We apply a small decrease, $\epsilon = 10^{-6}$, to its right-hand side to change this.

Constraint Set 4. *The linear constraints based on the schedulability conditions* (7)–(10) *are as follows.*

$$\begin{aligned}
&\forall p :: \hat{U}^A_{A,p} \leq 1 \\
&\forall p :: \hat{U}^B_{A,p} + \hat{U}^B_{B,p} \leq 1 \\
&\sum_{p=1}^{m} \left(\hat{U}^C_{A,p} + \hat{U}^C_{B,p} \right) + \hat{U}^C_C \leq m \\
&\sum_{p=1}^{m} \left(\hat{U}^C_{A,p} + \hat{U}^C_{B,p} \right) + (m-1)\hat{h} + \hat{H} \leq m - \epsilon
\end{aligned}$$

Linear program for LLC allocation. From Lemmas 1-3 and the discussion above, we have the following.

LP Allocation Theorem. *An allocation scheme that produces the minimum Level-C utilization for a task set while maintaining all schedulability conditions can be determined by solving the following LP.*

$$\text{minimize} \sum_{p=1}^{m} \left(\hat{U}_{A,p}^{C} + \hat{U}_{B,p}^{C} \right) + \hat{U}_{C}^{C}$$

subject to: Constraint Sets 1-4
Non-negativity constraints on all variables.

The objective of minimizing total Level-C utilization is used here as a greedy heuristic because this reduces tardiness bounds for Level-C tasks (Devi and Anderson 2008). However, this objective function serves a secondary purpose. Recall from our discussion of the LP variable $\hat{f}$ that if $\hat{f}$ is minimized, then it will equal $f(\hat{x})$ at integer values of $\hat{x}$. Minimizing total Level-C utilization ensures that utilization variables reflect actual system utilization values determined from PETs when LLC area variables are at integer values.

Approximations. Under certain scenarios, the LP above will converge to integer way values for many task systems. Consider the LP with Constraint Set 4 removed. The remaining constraints on Level-C utilizations from Constraint Set 2 intersect at integer way values. Level-C utilization is minimized at the intersection of linear constraints, and the LP will thus converge to integer values. However, the way-parameter values that minimize Level-C utilization may violate schedulability conditions (7), (8), or (10). In this scenario, the LP *with* Constraint Set 4 may not converge to integer way values.

If the program solution does not return integer values, we can round way values, or convert the LP to a mixed-integer LP (MILP). In prior work, we compared schedulability for rounded LP-based LLC allocation sizes to schedulability for MILP-based LLC allocation sizes (Chisholm et al. 2015). Note that non-integral LP-based LLC allocation sizes are not necessarily guaranteed to be nearest to integral LLC allocations that are schedulable when schedulable allocations exist. We found in our comparison that the schedulability loss due to rounded LP-based programming is fairly small in many cases. We also found that the time required to solve the MILP version was comparable to that required to solve the LP version. We therefore used the more-exact MILP version in the schedulability study considered next. However, we also considered the LP version in a subset of the experiments in this study to provide additional comparisons to the MILP version.

6 Evaluation

In Sec. 4, we analyzed the effect of different hardware-management choices on the performance of individual benchmark and micro-benchmark programs. In this section, we analyze the impact of different hardware-management and scheduling choices

on overall task-system schedulability. We conducted a large-scale overhead-aware schedulability study involving task systems randomly generated using a process based on our collected execution time data. A subset of these experimental results is presented in Sec. 6.1. In these schedulability experiments, certain provisioning assumptions were made regarding PETs. To partially assess the soundness of these assumptions, we conducted experiments in which runtime data was collected for systems of benchmark and micro-benchmark programs. This data is presented and discussed in Sec. 6.2.

6.1 Overhead-Aware Schedulability Experiments

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

- **MC^2-V1**: MC^2 under LLC-Allocation Variant 1 with DRAM bank isolation at Levels A and B.
- **MC^2-V2**: MC^2 under LLC-Allocation Variant 2 with DRAM bank isolation at Levels A and B. Differences in schedulability between MC^2-V1 and MC^2-V2 allow us to analyze the tradeoffs noted in Obs. 7 at Level C.
- **MC^2-V3**: MC^2 under LLC-Allocation Variant 1 with *no* DRAM bank isolation. Differences in schedulability between MC^2-V1 and MC^2-V3 allow us to analyze the extent to which LLC-isolation alone contributes to isolation benefits.
- **MC^2**: MC^2 with no DRAM bank or LLC isolation. This scheme provides the advantage of MC analysis only.
- **PEDF-ISO**: Partitioned EDF (PEDF) with DRAM bank and LLC isolation. This scheme provides the advantage of hardware-management only, thus all tasks use Level-A PETs. PEDF has been shown in previous work (Brandenburg 2011) to be perhaps the most competitive known HRT scheduling algorithm for multiprocessors when considering implementation overheads.
- **PEDF**: PEDF with no DRAM bank or LLC isolation.
- **EDF**: EDF on only one core. As stated in Sec. 1, this scheme represents the current industry best practice of disabling all but one core.

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

Schedulability experimental framework. In our schedulability experiments, we assumed that PETs for each criticality level are defined as in Sec. 2. Task sets were randomly generated by using five uniform distributions to choose task and task-set parameters. The specific distributions used were selected from the per-distribution choices listed in Table 2. These distributions are defined with respect to the single-core EDF scheme. All combinations of these choices were considered. These distributions determine the criticality utilization ratio (*i.e.* the fraction of the overall utilization assigned to each criticality level), task periods, task utilizations, the maximum LLC

Table 2 Task-set parameters and distributions.

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

reload time after a preemption or migration (specified as a fraction of overall task execution time), and per-task Level-A inflation factors (which are similar to those considered by Vestal (2007)).

At a high level, our overall experimental framework was as follows:

Step 1 Select the specific five distributions used from among the choices listed in Table 2.

Step 2 Generate task and task-set parameters under the single-core EDF scheme using these distributions.

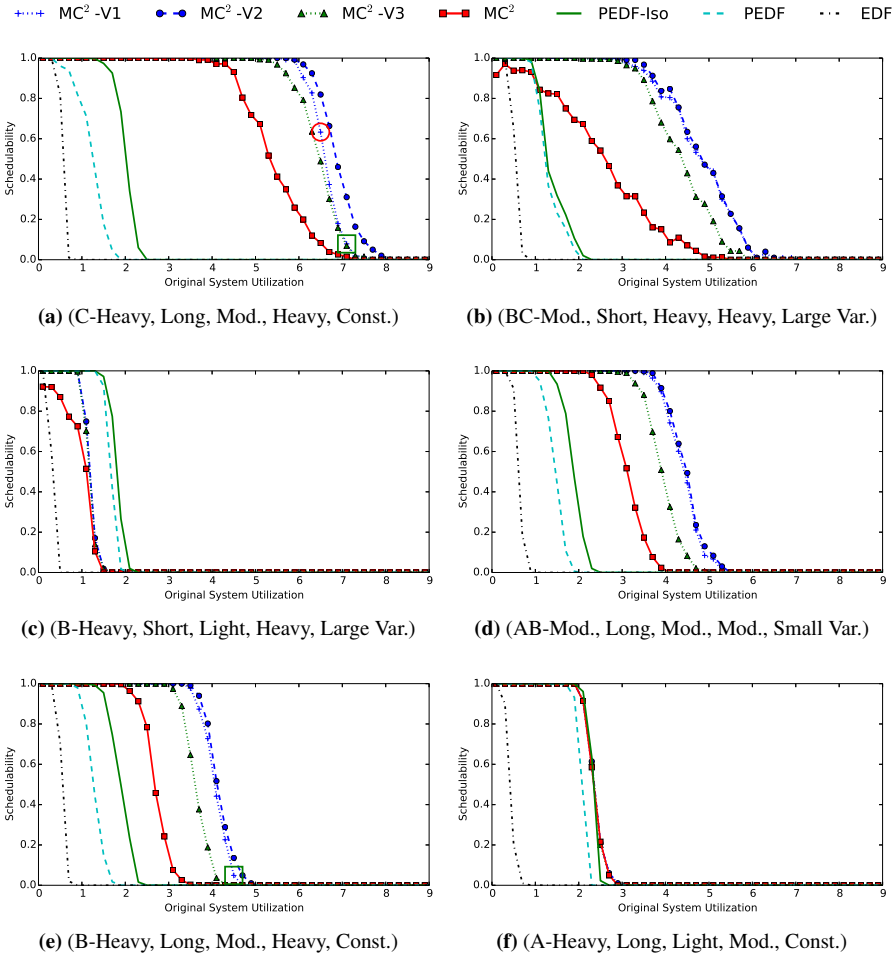
Step 3 Based on the generated EDF PETs, generate PETs for other isolation configurations and MC-provisioning assumptions (*e.g.*, PETs should be smaller in schemes that provide isolation compared to those that do not)—this step is described in greater detail in Appendix B.

Step 4 Adjust task parameters to account for relevant overheads—this step is described in greater detail in Appendix A.

Step 5 Check the schedulability of the resulting task set under each considered scheme.

In the third and fourth steps, the adjustments to apply were based upon measurement data. (We collected 8GB of task execution-time data and 9GB of overhead data.)

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



(a) (C-Heavy, Long, Mod., Heavy, Const.)

(b) (BC-Mod., Short, Heavy, Heavy, Large Var.)

(c) (B-Heavy, Short, Light, Heavy, Large Var.)

(d) (AB-Mod., Long, Mod., Mod., Small Var.)

(e) (B-Heavy, Long, Mod., Heavy, Const.)

(f) (A-Heavy, Long, Light, Mod., Const.)

Fig. 22 Representative schedulability plots. The points highlighted with a circle and squares are referenced in Secs. 6.1 and 6.2, respectively.

100 and 2,000 task sets, while ensuring that enough were generated to estimate the mean schedulability under a given combination to within ± 0.05 with 95% confidence.

For the schemes that support LLC isolation, we determined allocated LLC areas using the optimization methods in Sec. 5.3 with small variations where necessary. Under the MC^2-V2 scheme, we divided the overall Level-C area into fourths (rounding as necessary) to give per-core areas. For schemes requiring task partitioning, we used the worst-fit-decreasing bin-packing heuristic.

Schedulability results. In total, we evaluated the schedulability of approximately three million randomly generated task sets, which took roughly 18 CPU-days of computation. From this abundance of data, we generated over 500 schedulability plots,

of which six representative plots are shown in Fig. 22. The full set of plots is available online at `http://www.cs.unc.edu/~anderson/papers.html`.

Each schedulability plot corresponds to a specific task-set category corresponding to a specific combination of the parameter distributions in Table 2. To understand how these plots are interpreted, consider Fig. 22(a). For this plot's task-set category, the circled point indicates that 63% of the generated task sets with EDF utilizations of 6.5 were schedulable under MC^2 with criticality-cognizant isolation. Note that, because the x-axis represents system utilizations under the single-core HRT EDF scheme, it is possible under MC^2 to support systems with an EDF utilization exceeding four, as the MC-provisioning assumptions decrease PETs.

We now state several observations that follow from the full set of collected schedulability data. We illustrate these observations using the data presented in Fig. 22.

Obs. 11 *MC^2 schemes were able to schedule at least one more core's worth of utilization in comparison to the PEDF schemes in 64% of considered scenarios. Within this 64%, the MC^2 schemes were able to schedule 1.8 to 2.8 cores' worth of additional utilization on average.*

This observation is supported by insets (a), (b), (e) and (f) of Fig. 22. The scenario in inset (a) corresponds to the industry-inspired motivation underlying the specification of MC^2 (see Sec. 2), as in this scenario, only tasks of rather light utilizations exist at Levels A and B. For some scenarios where loads are concentrated at higher criticality levels, the MC^2 schemes also yielded substantial schedulability improvements. Insets (e) and (f) provide examples.

Obs. 12 *LLC and DRAM bank isolation allowed PEDF to only schedule 0.26 cores' worth of additional utilization on average. However, MC^2 was able to schedule 0.93 cores' worth of additional utilization on average under LLC and DRAM bank isolation, sometimes scheduling two, and occasionally almost four, cores' worth of additional utilization.*

Insets (e) and (f) of Fig. 22 give two examples of this for A-heavy and B-heavy task sets. In the MC^2-V1 scheme, high-criticality tasks are afforded the benefits of the LLC while the unmanaged cases are not. However, isolation yields little improvement under PEDF. This is because, under PEDF-ISO, *all* tasks are assigned to cores and contend for LLC allocations, while in MC^2, Level-C tasks share LLC space apart from higher criticality levels. This shows that criticality-cognizant isolation can provide major schedulability benefits not seen in criticality-oblivious isolation.

Obs. 13 *PEDF scheduled more task sets than MC^2 in 22% of the considered scenarios, particularly those most affected by overheads (short periods, light-utilization tasks, or both). In 9% of scenarios, schedulability under MC^2 and PEDF differed by less than 25%.*

Fig. 22(c) presents a scenario for which PEDF outperformed unmanaged MC^2 at certain utilizations, due to additional overheads for MC^2. Fig. 22(d) presents a scenario for which most schemes have nearly equivalent schedulability performance. However, for most scenarios, the benefits of MC provisioning outweighed the disadvantages of additional overheads under MC^2.

Obs. 14 *Adding LLC isolation for Level C in MC^2 resulted in less than 5% difference between MC^2-V1 and MC^2-V2 schedulability in 98% of the considered scenarios and resulted in a 3–7% gain in schedulability under MC^2-V2 in 10% of scenarios.*

In the specific scenarios in Fig. 22, the impact of adding Level-C LLC isolation ranges from negligible (inset (c)) to slight (inset (e)) to moderate (inset (a)).

Obs. 15 *Under MC^2-V3, schedulability gains from LLC isolation were 75% of the gains from combining LLC isolation and DRAM bank isolation under MC^2-V1.*

Insets (d) and (e) of Fig. 22 show cases where DRAM bank isolation provided significant increases to the schedulability benefits of isolation, as determined by comparing the MC^2, MC^2-V1, and MC^2-V3 curves. In contrast, insets (a) and (b) show cases where DRAM bank isolation contributed very little to isolation-related schedulability gains. While DRAM bank isolation provided notable benefits in some cases, LLC isolation was the predominant contributor to isolation-related schedulability benefits under MC^2-V1.

Obs. 16 *Under the MC^2-V1 scheme, the LP and MILP methods for determining LLC allocations yielded similar schedulability results and required similar computing times.*

We re-ran the schedulability experiments for the MC^2-V1 scheme and evaluated both LP- and MILP-derived LLC allocations. Across all task sets, the LP method took 0.338 seconds to run on average and 5.6 seconds in the worst case, and the MILP method took 0.343 seconds to run on average and 120.3 seconds in the worst case. The difference in schedulability under these two schemes, as determined by comparing the area under each schedulability curve, was less than 0.01% across all categories. In extremely rare cases (less than 0.001% of all task sets evaluated), the MILP method's computing time spiked. If the MILP method ran longer than 2 minutes on a given task set, we terminated it and deemed the task set not schedulable. The MILP method's maximum runtime over all samples not timed out was 3.1 seconds.

The schedulability study presented shows that, in many cases, MC^2 and hardware management significantly improve schedulability performance when combined. However, the conclusions drawn from this study are predicated on our provisioning assumptions, in particular the assumption that tasks can be reasonably provisioned based on *measured* WCETs and ACETs. Sec. 6.2 presents our investigation into the safety of this assumption.

6.2 Case Studies

To investigate the validity of our MC^2 provisioning assumptions, we created ten task systems for the scenarios in insets (a) and (e) of Fig. 22 corresponding to the highest-utilization point in the MC^2-V1 and MC^2-V2 curves with non-zero schedulability. (The scenarios of these two insets represent interesting use cases for MC^2-V1 and MC^2-V2, given that schedulability under MC^2-V1 or MC^2-V2 is significantly greater than schedulability under other schemes.) The point chosen for MC^2-V1 is highlighted

by a square in each figure. Each task system was composed of synthetic micro-benchmark programs for Levels A and B, since they are quite deterministic and have smaller WSSs, and DIS benchmark programs for Level C. Table 3 gives an example of one of these task systems. Each of these task systems was executed for ten minutes each.

Across all of these task systems, there were no Level-A or -B deadline misses. At Level C, there were deadline misses—recall this is acceptable at Level C—but of the Level-C tasks, the largest relative deadline miss (response time divided by period) under MC^2-V1 (respectively, MC^2-V2) was 1.35 (respectively, 1.02), and the largest deadline-miss ratio under MC^2-V1 (respectively, MC^2-V2) was 1.40% (respectively, 0.06%). These results are likely acceptable for most SRT applications. These results suggest that our provisioning assumptions are reasonable, and support our analytical schedulability results.

7 Conclusion

We presented a significant extension to the MC^2 framework that provides LLC and DRAM-bank isolation and that isolates the OS from high-criticality tasks. We also presented an optimization framework for determining LLC allocations beneficial to MC^2 schedulabillity. Additionally, we conducted extensive experiments (with substantially more data found online (Kim et al. 2016c)) in which the impact of the newly provided isolation mechanisms was assessed individually as well as collectively from a system-wide schedulability point of view. To our knowledge, this is the first work to explore criticality-cognizant hardware-management techniques with the goal of improving platform utilization, the first work that considers isolating the OS from application-level real-time tasks, the first work to optimize LLC allocations in the context of MC systems, and the first work to show that the one-out-of-m problem can be effectively addressed through criticality-cognizant hardware management.

This paper suggests many avenues for future work. While we have considered a vast array of possible LLC and DRAM-bank configurations, we made several assumptions to keep the design space tractable. For example, we assumed an even distribution of colors among cores at Levels A and B is preferable for LLC allocation variants that provide isolation at these levels. Other allocation strategies could potentially yield improved results. We also assumed that Level-C tasks were provisioned based on ACETs. While our case studies suggest this assumption yields positive results, in the future we plan to explore alternative provisioning assumptions and the tradeoff between increased platform utilization and deadline tardiness and miss ratios. For example, if we provision Level-C tasks based on the 70$^{\text{th}}$ percentile, how much smaller would observed tardiness be? Regarding our implementation itself, the most pressing concern is further extensions to handle shared data. As mentioned earlier, some work in this direction has already been conducted (Chisholm et al. 2016; Kim et al. 2016a).

This paper builds upon two prior conference papers (Chisholm et al. 2015; Kim et al. 2016b). The main additions are: **(i)** we evaluated all possible combinations of LLC and DRAM bank isolation techniques; **(ii)** we examined and discussed unexpected results in a subset of the micro-benchmark experiments that were performed

Table 3 An example of the generated task system.

Task ID (program)	Assigned CPU	Criticality Level	e_i^A (ms)	e_i^B (ms)	e_i^C (ms)	T_i (ms)
τ_1^A (micro-benchmark)	0	A	7.78	5.19	4.84	96
τ_2^A (micro-benchmark)	1	A	2.23	1.49	1.30	48
τ_3^A (micro-benchmark)	2	A	5.02	3.35	3.12	48
τ_4^A (micro-benchmark)	3	A	4.71	3.14	2.93	48
τ_5^A (micro-benchmark)	1	A	1.64	1.09	0.90	48
τ_6^A (micro-benchmark)	1	A	4.81	3.21	2.99	96
τ_7^A (micro-benchmark)	0	A	4.28	2.85	2.66	48
τ_8^A (micro-benchmark)	3	A	2.35	1.57	1.39	48
τ_9^A (micro-benchmark)	2	A	7.09	4.73	4.41	96
τ_{10}^A (micro-benchmark)	1	A	5.09	3.34	3.12	96
τ_{11}^B (micro-benchmark)	3	B	-	5.95	5.59	192
τ_{12}^B (micro-benchmark)	0	B	-	22.40	20.64	96
τ_{13}^B (micro-benchmark)	2	B	-	6.47	6.10	192
τ_{14}^B (micro-benchmark)	1	B	-	10.71	10.20	96
τ_{15}^B (micro-benchmark)	3	B	-	9.55	9.14	192
τ_{16}^B (micro-benchmark)	2	B	-	16.44	15.37	96
τ_{17}^B (micro-benchmark)	3	B	-	4.28	4.00	192
τ_{18}^B (micro-benchmark)	0	B	-	17.80	16.60	96
τ_{19}^B (micro-benchmark)	1	B	-	19.34	17.94	192
τ_{20}^B (micro-benchmark)	3	B	-	8.42	8.03	192
τ_{21}^B (micro-benchmark)	2	B	-	22.75	20.94	96
τ_{22}^B (micro-benchmark)	1	B	-	4.78	4.46	192
τ_{23}^B (micro-benchmark)	3	B	-	13.33	12.56	96
τ_{24}^B (micro-benchmark)	0	B	-	13.88	13.06	96
τ_{25}^B (micro-benchmark)	1	B	-	12.49	11.80	192
τ_{26}^B (micro-benchmark)	2	B	-	9.51	9.10	96
τ_{27}^B (micro-benchmark)	1	B	-	9.52	9.12	96
τ_{28}^B (micro-benchmark)	3	B	-	10.76	10.25	192
τ_{29}^B (micro-benchmark)	0	B	-	20.46	19.01	96
τ_{30}^B (micro-benchmark)	3	B	-	11.51	10.92	192
τ_{31}^C (Update)	-	C	-	-	5.85	125.40
τ_{32}^C (Update)	-	C	-	-	5.85	47.88
τ_{33}^C (Field)	-	C	-	-	9.16	219.80

(see Obs. 5 and Fig. 9 in Sec. 4 and associated discussion); **(iii)** we evaluated cross-core OS interference (see Fig. 15 in Sec. 4 and associated discussion); **(iv)** we considered a new LLC-allocation strategy (see discussion of Fig. 6 (c)).

Acknowledgements We are grateful to Cheng-Yang Fu for assisting with some of the implementation efforts discussed in this paper.

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 317–327

Alhammad A, Wasly S, Pellizzoni R (2015) Memory efficient global scheduling of real-time tasks. In: Proceedings of the 21th 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

Audsley N (2013) Memory architecture for NoC-based real-time mixed criticality systems. In: Proceedings of the 1st International Workshop on Mixed Criticality Systems, pp 37–42

Baker T, Shaw A (1988) The cyclic executive model and ada. In: Proceedings of the 9th IEEE Real-Time Systems Symposium, pp 120–129

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

Bui B, Caccamo M, Sha L, Martinez J (2008) Impact of cache partitioning on multi-tasking real time embedded systems. In: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 101–110

Burns A, Davis R (2016) Mixed criticality systems – a review. Tech. rep., Department of Computer Science, University of York

Campoy M, Ivars A, Mataix J (2001) Static use of locking caches in multitask preemptive real-time systems. In: Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop, pp 1283–1286

Certification Authorities Software Team (CAST) (2014) Position paper CAST-32: Multi-core processors. `https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf`

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 Real-Time Systems Symposium, pp 305–316

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

Devi U, Anderson J (2008) Tardiness bounds under global EDF scheduling on a multiprocessor. Real-Time Systems 38:133–189

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

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 73–83

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 21th 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

Jalle J, Quinones E, Abella J, Fossati L, Zulianello M, Cazorla F (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

Kessler R, Hill M (1992) Page placement algorithms for large real-indexed caches. ACM Trans Comput Syst 10:338–359

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, Niz DD, Andersson B, Klein M, Mutlu O, Rajkumar R (2014) 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 21th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 317–326

Kim N, Chisholm M, Otterness N, Anderson J, Smith F (2016a) Allowing shared libraries while supporting hardware isolation in multicore real-time systems. Manuscript, available at `http://www.cs.unc.edu/~anderson/papers.html`

Kim N, Ward B, Chisholm M, Fu C, Anderson J, Smith F (2016b) Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In: Proceedings of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium, pp 149–160

Kim N, Ward B, Chisholm M, Fu C, Anderson J, Smith F (2016c) Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning (full version). Available at `http://www.cs.unc.edu/~anderson/papers.html`

Kirk D (1989) SMART (strategic memory allocation for real-time) cache design. In: Proceedings of the 10th IEEE Real-Time Systems Symposium, pp 229–237

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 International Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems

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

Kroft D (1981) Lockup-free instruction fetch/prefetch cache organization. In: Proceedings of the 8th Annual Symposium on Computer Architecture, pp 81–87

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

Mills A, Anderson J (2011) A multiprocessor server-based scheduler for soft real-time tasks with stochastic execution demand. In: Proceedings of the 17th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp 207–217

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 Conferences on Embedded Software and Systems, pp 1864–1871

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

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

Tabish R, Mancuso R, Wasly S, Alhammad A, Phatak S, Pellizzoni R, Caccamo M (2016) 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 161–172

Vestal S (2007) Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In: Proceedings of the 28th IEEE 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 123–134

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 multicoore platforms. In: Proceedings of the

20th IEEE Real-Time and Embedded Technology and Applications Symposium, pp 155–166

A Overhead Accounting

In our schedulability study, to account for implementation-related overheads beyond those discussed in Sec. 5.2, we applied several existing overhead-accounting techniques (Brandenburg 2011). While a complete, formal description of all techniques is beyond the scope of this paper, in what follows, we give a high-level description of the techniques employed, and highlight the most relevant ideas. We account for all overhead sources through PET *inflation*, *i.e.*, increasing the PET of each task before evaluating schedulability. In addition to CRPDs, we considered the following overhead sources, defined in Brandenburg (2011): context switching, release latency, timer ticks, scheduling, job release, and inter-processor interrupts (IPIs).

To account for these overheads, we applied techniques pioneered by Brandenburg (2011) for PEDF and GEDF, for Level B and Level C, respectively, with minor modifications to account for interactions among criticality levels in MC^2. When analyzing tasks at each criticality level, we used measured overheads acquired using similar assumptions as used for PETs, *e.g.*, at Level C, average-case measured overheads were considered. Also, in the case of scheduling and release overheads, we have to account for per-core partitioned scheduling and release overheads at Levels A and B, and also global scheduling and release overheads that may be incurred on any core for Level C. Releases and IPI overheads from task migrations at Level C may cause delays at all criticality levels.

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

B PET-Generation Process

The PETs assumed in Sec. 6.1 are based on an analytical model, which we derived by distilling the measured execution-time data discussed in Sec. 4. This appendix describes this PET-generation model in greater detail. As described in Sec. 6.1, all PETs required in our schedulability experiments are defined based on EDF-scheme PETs, which correspond to A-inflated WCETs in an idle system with the full LLC allocated to the task in question. We denote this WCET parameter as C_i^0 for task τ_i. In our experimental framework, the C_i^0 values are obtained implicitly from the randomly generated task utilizations and periods. All execution-time values used to obtain all other PETs for τ_i for different isolation and analysis assumptions are listed in Table 4. Table 5 shows how these values are used to define all PETs under each scheme. The columns of Table 4 indicate how each execution-time value is defined (*i.e.*, whether the value is a Level-A-inflated WCET, a non-inflated WCET, or an ACET, whether the system is assumed to be under load or idle, *etc.*). Each of these values is generated by applying scaling factor(s) to the prior-listed execution-time values. We present an overview of this entire process here.

Table 4 Generated PET values.

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

Table 5 Assignment of execution time parameters to PETs.

Task Level	PET Level	MC^2 -V1	MC^2 -V2	MC^2	PEDF -ISO	PEDF	EDF
A	A/HRT	C_i^2	C_i^2	C_i^4	C_i^2	C_i^4	C_i^0
	B	C_i^5	C_i^5	C_i^5	N/A	N/A	N/A
	C	C_i^6	C_i^6	C_i^8	N/A	N/A	N/A
B	A/HRT	N/A	N/A	N/A	C_i^2	C_i^4	C_i^0
	B	C_i^5	C_i^5	C_i^5	N/A	N/A	N/A
	C	C_i^6	C_i^6	C_i^8	N/A	N/A	N/A
C	A/HRT	N/A	N/A	N/A	C_i^2	C_i^4	C_i^0
	B	N/A	N/A	N/A	N/A	N/A	N/A
	C	C_i^8	C_i^7	C_i^8	N/A	N/A	N/A

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

Step 2: Generate C_i^2 by scaling C_i^1 for different LLC allocations. Our C_i^0 values are defined from generated utilizations. The process for generating such utilizations was carefully defined to produce trends similar to those seen from measurement data. Since our C_i^1 values are simply scaled versions of our C_i^0 parameters, similar utilization trends will be seen when utilizations are defined in terms of C_i^1 values. Fig. 23 illustrates typical generated utilizations. As seen in this figure, task utilizations monotonically decrease with increasing LLC space and converge at the ICAS. This is in accordance with Obs. 2. To reflect this, we obtain C_i^2 values for different LLC-allocation choices by applying a scaling factor to C_i^1 that exponentially increases with

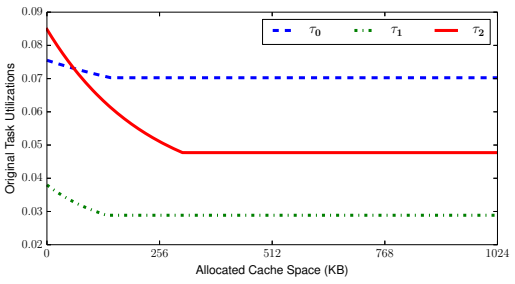


Fig. 23 Utilizations generated under different LLC allocations for three example tasks.

the minimum of the ICAS and LLC space. The actual scaling factors employed were selected to reflect measurement data.

Task ICASs were deduced using the Load Time parameter in Table 2. The two both hinge on a task's cache footprint. Our Load Time parameter was defined to reflect Obs. 1, which showed that cache isolation can improve a task's WCET by up to 369%. For example, when the Light Load Time distribution is assumed, LLC isolation typically reduces WCETs by 20-50%, while when the Heavy distribution is assumed, the reduction is typically 200-500%. In addition, for all parameter combinations, tasks at Levels A and B tend to be more insensitive to LLC space than those at Level C. This reflects the underlying motivation for MC^2 that Level-A and -B tasks will tend to be rather deterministic fly-weight tasks and that Level-C tasks will tend to be more complex data-intensive tasks (see Sec. 2).

Step 3: Generate C_i^3 by scaling C_i^2 to account for shared DRAM banks. As seen in Fig. 7, the impact of DRAM bank isolation on task execution times tended to range from imperceptible to 20% under small LLC allocations. Based on these results, we uniformly choose C_i^3 to be $[100, 130)\%$ of C_i^2 to account for the lack of bank isolation. Similar to the last step, this step is affected by the task ICAS and LLC allocation.

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

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

Step 6: Generate C_i^6 and C_i^7 to reflect expected ACET:WCET ratios under cache isolation and varying background workloads. ACET:WCET ratio trends will depend on the given background workload, *i.e.*, the total utilization of all competing tasks. Based on ACET:WCET ratio trends observed for benchmark and synthetic programs under different background workload utilizations, we identified an appropriate distribution from which to uniformly choose an ACET:WCET ratio for each task. For all tasks, these ratios were chosen uniformly among a range of percentages. For Level-C tasks, these ratios range over 20-40% for the lightest background workloads, and over 30-60% for the heaviest. For Level-A and -B tasks, these ratios range over 50-70% for the lightest background workloads, and 80-100% for the heaviest. This reflects our assumption that higher-criticality tasks tend to be more deterministic in their execution than Level-C tasks. Note that this process requires a means to calculate the Level-C utilization of a background-workload, which is dependent on the ACETs that are generated. This entails an iterative process, such as that in Fig. 24(a). However, given the scale of our schedulability experiments, which involved millions of task systems, an iterative process was infeasible. As a result, we used the non-iterative process outlined in Fig. 24(b). We used the EDF-scheme utilization of the background workload as an upper bound on its average-case utilization.

Step 7: Generate C_i^8 to reflect differences between ACETs for a fully unmanaged system and ACETs for a cache-isolated system. From Fig. 7 and Obs. 8,

Step 6.1 Assign all tasks Level-C PETs based on ACETs in an idle system.

Step 6.2 Based on assigned Level-C PETs, calculate the Level-C utilization of the background workload of each task.

Step 6.3 Assign new Level-C PETs based on ACETs under the background-workload utilizations calculated in Step 6.2.

Step 6.4 Repeat Step 6.2 and 6.3 if any PETs increased in Step 6.3.

(a) Iterative process.

Step 6.1 Based on the assigned EDF-scheme PET for each task, calculate the EDF-scheme utilization of the background workload of each task.

Step 6.2 Assign Level-C PETs based on ACETs under the background workload utilizations calculated in Step 6.1.

(b) Non-iterative process.

Fig. 24 Two methods for calculating ACETs in Step 6.

we see that ACETs in an unmanaged cache gradually decline in a linear fashion as the allocated LLC space increases, even beyond the ICAS of the task. However, these ACETS generally remain higher than ACETs under cache isolation. When the LLC allocation is zero, both ACETs are the same, since LLC management does not affect tasks bypassing the LLC. To reflect this behavior, we generated C_i^8 as shown in Fig. 25. On the right axis, we depict a scale showing the range of C_i^7's reduction in value as the allocated LLC space increases. On this scale, C_i^7 is at 0% reduction under zero allocated LLC space, and 100% under maximum allocated LLC space. C_i^8 at maximum allocated LLC space for the Matrix program would fall at approximately 50% on this scale. For each generated task, we choose a value from 30-70% on this scale for our generated C_i^8 at maximum LLC space. At zero allocated LLC space, C_i^8 matches C_i^7. For all other LLC allocation sizes, we interpolate values for C_i^8 linearly between values generated for zero allocated LLC space and maximum allocated LLC space.

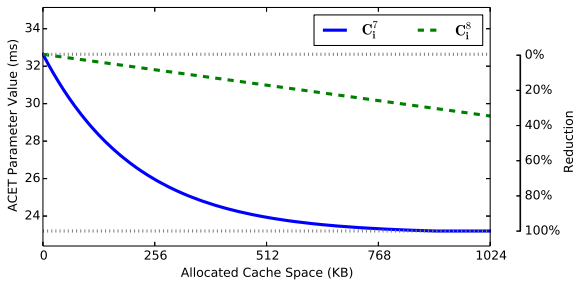


Fig. 25 Comparison of C_i^7 and C_i^8 for a generated task.

From these steps, we now have all required PETs. We note once again that this process produces a *model* for producing PETs. As such, all claims resulting from our schedulability experiments apply only within the context provided by this

model. Still, we have taken great pains to ensure that the range of PETs generated by this model encompass those that we have seen based on real measurement data, and that trends among related PETs for the same task correspond to those seen in our measurement data.