Supporting Mode Changes while Providing Hardware Isolation in Mixed-Criticality Multicore Systems [*]

Micaiah Chisholm, Namhoon Kim, Stephen Tang, Nathan Otterness,
James H. Anderson, F. Donelson Smith, and Donald Porter
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

When hosting real-time applications on multicore platforms, *interference* from shared hardware resources (*e.g.* caches and memory banks) can significantly increase task execution times. Most proposed approaches for lessening interference rely on mechanisms for providing *hardware isolation* to tasks. However, one limitation of most prior work on such mechanisms is that only static task systems have been considered that never change at runtime. In reality, safety-critical applications often transition among different functional *modes*, each defined by a distinct set of running tasks. In a given mode, only tasks from that mode execute, yet tasks from all modes consume memory space, and this creates additional constraints affecting hardware-isolation techniques. This paper shows how to address such constraints in the context of an existing real-time resource-allocation framework called MC^2 (mixed-criticality on multicore). In MC^2, hardware-isolation techniques are employed in conjunction with criticality-aware task-provisioning assumptions that enable hardware resources to be utilized more efficiently.

1 Introduction

There is great interest today in hosting computationally intensive real-time workloads on multicore platforms. However, efforts towards this end have been stymied by problems caused when tasks interfere with each other in accessing shared hardware components such as caches and memory banks. Shared-hardware interference can cause significant task execution-time increases. This is especially problematic for real-time workloads, which are typically validated by analyzing worst-case scenarios. When worst-case execution times increase proportionally to the amount of sharing across cores, the benefit of additional cores can be nullified. This dilemma has been dubbed *the one-out-of-m problem* [24] to reflect the very real possibility of being able to allocate only "one core's worth" of capacity though m cores are present.

The one-out-of-m problem is one of the most serious unresolved obstacles in work on real-time multicore resource allocation today. Evidence of this can be seen in the recent CAST-32 position paper from the U.S. Federal Aviation Administration (FAA) [7, 8]. This position paper provides an in-depth discussion of the challenges created by employing multicore platforms in avionics settings.

In addressing the one-out-of-m problem, two orthogonal approaches have been investigated. The predominate approach involves devising mechanisms to predictably manage shared hardware resources so that interference and task execution-time estimates are reduced [1, 2, 3, 4, 9, 12, 13, 14, 15, 17, 18, 19, 20, 21, 23, 24, 26, 27, 29, 32, 34, 37, 39, 38, 40, 41]. Alternatively, Vestal (while working in the avionics industry) proposed employing *mixed-criticality* (*MC*) analysis [36], under which execution-time estimates for less-critical tasks are determined based on less-pessimistic assumptions.[1] While these two approaches have been largely considered separately, both have been applied together in ongoing work by our group on a real-time resource-allocation framework called MC^2 (mixed-criticality on multicore) [9, 16, 23, 24, 29, 37]. In particular, MC^2 supports both MC provisioning techniques as proposed by Vestal [36] and mechanisms for managing the shared last-level cache (LLC) and DRAM memory.

From static to dynamic workloads. Prior work on shared-hardware management has been almost entirely limited to static task systems that never change at runtime. This is a key limitation. Indeed, many safety-critical applications must support multiple functional *modes*, each defined by a distinct set of running tasks. For example, in an aircraft, different sets of running tasks may be required when taking off, at cruise altitude, or when some emergency condition occurs. Thus, for the one-out-of-m problem to be truly solved, it is crucial that solutions exist that encompass multi-mode systems. In this paper, we present such a solution in the context of MC^2.

Allowing multiple modes to exist can greatly complicate shared-hardware management. The key issue here is not exhausting overall CPU capacity, because tasks from different modes do not run at the same time. Rather, in the context of MC^2, DRAM allocations are the main problem, as even inactive tasks consume memory. Note that the DRAM region a task can access determines the region of the LCC it accesses.[2] Thus, the problems of allocating DRAM space and LLC space are intertwined and many complexities exist.

Contributions. MC^2 includes an offline DRAM/LLC allocation component, and an online component that schedules tasks at runtime. Our major contribution is to show how to modify both components to support multi-mode systems. We also report on the results of experiments conducted using our modified MC^2 in which various shared-hardware allocation

[*]Work supported by NSF grants CNS 1409175, CPS 1446631, and CNS 1563845, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

[1]Vestal originally considered uniprocessor platforms, but MC analysis has also been considered in the context of multicore platforms.

[2]The LLC would typically be a set-associative cache with the mapping of a memory location to a set determined by its physical address.

options pertaining to such systems were explored.

MC^2's offline component allocates DRAM and LLC regions to subsets of tasks in a criticality-cognizant way. When modifying this component to support multi-mode systems, different regions of DRAM and the LLC must be assigned to *per-mode* subsets of tasks while ensuring that all tasks from *all modes* are so assigned and *each mode* is schedulable.

With respect to these requirements, tasks that are *shared* across multiple modes (as commonly occurs in practice) can cause major difficulties because their presence creates dependencies among modes. To further complicate matters, a shared task could potentially be of different criticalities in different modes. For example, a planning computation in an unmanned aerial vehicle may require a criticality-level upgrade during a mode switch initiated in response to a detected threat, as planning becomes very critical in that context. To our knowledge, such *criticality changes* (under Vestal's notion of criticality [36]) have not been considered in prior work on mode changes.[3] A task that undergoes a criticality change when switching between two modes may require different hardware isolation guarantees in each mode.

When allocating DRAM to a task τ_i that is shared between two modes, two basic options exist: either the two modes can be allocated overlapping DRAM regions, with τ_i allocated in the overlap, or they can be allocated non-overlapping DRAM regions, which would entail migrating τ_i's state between these regions when switching between the two modes. Under either option, LLC allocations would be correspondingly affected. In more complex situations, these options could actually be applied in disparate combinations, with different techniques used for different tasks or modes. Along with other details pertaining to how DRAM and LLC regions are actually created, this yields a vast solution space to explore.

To sift through this solution space, we conducted a large-scale schedulability study in which overheads that impact schedulability were considered as measured on our multi-mode extension of MC^2's runtime component. We also conducted case-study experiments to confirm that mode-change latencies in our MC^2 extension are reasonable. To our knowledge, this paper is the first work on supporting mode changes in a multicore context where hardware-isolation and MC-analysis techniques (as proposed by Vestal [36]) are used.[4]

Organization. In the rest of this paper, we provide relevant background (Sec. 2), describe our modifications to MC^2 to support mode changes (Sec. 3), discuss our schedulability experiments (Sec. 4) and related work (Sec. 5), and conclude (Sec. 6). Due to space constraints, some implementation details and experimental results are deferred to appendices.

[3]Under current avionics certification procedures, such a task would always default to its highest criticality level. However, as noted in the CAST-32 position paper [7, 8], such procedures assume uniprocessor machines and must evolve to enable better platform utilization on multicore platforms. Moreover, according to colleagues in the avionics industry, many practical use cases exist where criticality changes would be desirable to support.

[4]As noted in Sec. 5, in prior work on MC analysis, a switch to degraded system performance is often cast as a mode change, but this is very different from the functional mode changes considered in this paper.

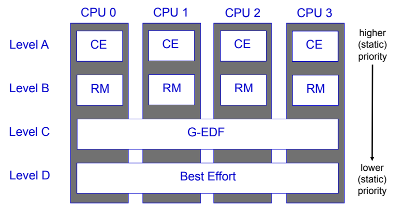


Figure 1: Scheduling in MC^2 on a quad-core machine.

2 Background

We begin by reviewing needed background material.

Task model. We consider real-time workloads specified via the implicit-deadline periodic/sporadic task model (of which we assume familiarity). We specifically consider a task system $\tau = \{\tau_1, \ldots, \tau_n\}$, scheduled on m processors,[5] where task τ_i's *period* and *worst-case execution time* (*WCET*) are denoted T_i and C_i, respectively. (We generalize this model below when considering MC scheduling and multi-mode systems.) The *utilization* of task τ_i is given by $u_i = C_i/T_i$ and the *total system utilization* by $\sum_i u_i$. If a job of τ_i has a deadline at time d and completes execution at time t, then its *tardiness* is $max\{0, t-d\}$. Tardiness should always be zero for a hard real-time (HRT) task, and should be bounded by a (reasonably small) constant for a soft real-time (SRT) task.

Mixed-criticality scheduling. For systems with tasks of differing criticalities, Vestal proposed using less-pessimistic execution-time estimates when considering less-critical tasks [36]. Under his proposal, if L criticality levels exist, then each task has a *provisioned execution time* (*PET*) specified at each level, and L system variants are analyzed: in the Level-ℓ variant, the real-time requirements of all Level-ℓ tasks are verified with Level-ℓ PETs assumed for *all* tasks (at any level). The degree of pessimism in determining PETs is level-dependent: if Level ℓ is of higher criticality than Level ℓ', then Level-ℓ PETs will generally exceed Level-ℓ' PETs. For example, in the systems considered by Vestal [36], observed WCETs were used to determine lower-level PETs, and such times were inflated to determine higher-level PETs.

Scheduling under MC^2. Vestal's work led to a significant body of follow-up work on MC scheduling (see [6] for an excellent survey). Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors [29]. MC^2 is implemented as a LITMUS$^{\mathrm{RT}}$ [28] plugin and supports four criticality levels, denoted A (highest) through D (lowest), as shown in Fig. 1. Higher-criticality tasks are statically prioritized over lower-criticality ones. Level-A tasks are periodic and partitioned and scheduled on each core using a time-triggered table-driven cyclic executive.[6] Level-B tasks are also periodic and 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 have harmonic periods and commence execution at time 0. Level-C tasks are

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

[6]Other per-level schedulers optionally can be used. These options, and other considerations, such as slack reallocation, schedulability conditions, and execution-time budgeting are discussed in prior papers [16, 29, 37].

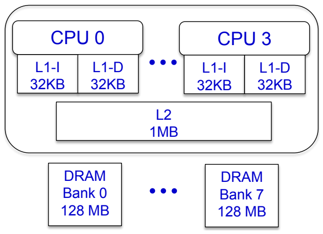


Figure 2: Quad-core ARM Cortex A9.

sporadic and 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. A major thesis underlying the design of MC^2 is that Levels A and B should be mostly comprised of quite deterministic "fly-weight" tasks with rather low utilizations; more computationally intensive tasks would likely be assigned to Level C. This thesis arose from discussions with colleagues in the avionics industry, who are interested in deploying complex decision-making capabilities at lower criticality levels.

Hardware management under MC^2. MC^2 provides a component used offline (*i.e.*, before runtime) to perform LLC and DRAM allocations [24]. We briefly describe the techniques that underlie this component here. (We are still assuming there is only a single task set to schedule. We consider extensions to support multiple modes later.) Our description is with respect to the machine shown in Fig. 2, which is the hardware platform assumed throughout this paper. This machine is a quad-core ARM Cortex A9 platform. Each core on this machine is clocked at 800MHz and has separate 32KB L1 instruction and data caches. The LLC is a shared, unified 1MB 16-way set-associative L2 cache. The LLC write policy is write-back with write-allocate. 1GB of off-chip DRAM is available, partitioned into eight 128MB banks.

We assume herein that Level D is not present, as it has no impact on the isolation guarantees or schedulability of tasks at higher levels (and Level D is afforded no real-time guarantees). LLC management is provided for the other levels by assigning rectangular areas of the LLC to certain groups of tasks. This is done by using *page coloring* to allocate certain subsequences of sets (*i.e.*, rows) of the LLC to such a task group, and hardware support in the form of per-CPU *lockdown registers* to assign certain ways (*i.e.*, columns) of the LLC to the group. (See [24] for more details concerning these LLC allocation mechanisms.) Also, by controlling the memory pages assigned to each task, certain DRAM banks can be assigned for the exclusive use of a specified group of tasks. The operating system (OS) can also be constrained to access only certain LLC areas or DRAM banks.

Fig. 3 depicts the main allocation strategy for the LLC and DRAM banks provided under MC^2 [24]. DRAM allocations are depicted at the bottom of the figure, and LLC allocations at the top. As seen, the Level-A and -B tasks on each CPU are assigned a dedicated DRAM bank, and Level C and the OS share the remaining banks. Also, Level C and the OS share a subsequence of the available LLC ways and all LLC colors. (On the considered platform, each color corresponds to 128 cache sets.) Level-C tasks (being SRT) are assumed to be provisioned on an average-case basis. Accordingly, 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 one quarter of the available colors. This scheme ensures that Level-A and -B tasks do not experience LLC interference from tasks on other cores (*spatial* isolation). Also, Level-A tasks (having higher priority) do not experience LLC interference from Level-B tasks on the same core (*temporal* isolation).

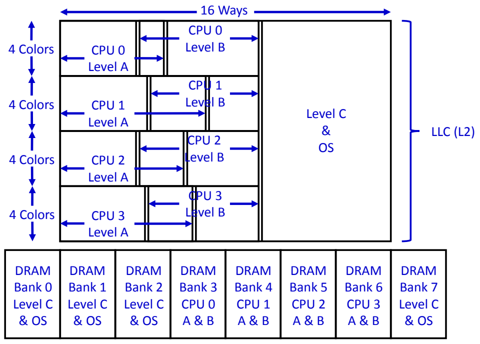


Figure 3: LLC and DRAM bank allocation. Note that the Level-A and -B LLC areas for each core can overlap. LLC boundaries indicated by double lines are configurable parameters.

For any task set, the actual number of ways allocated to each LLC partition (*i.e.*, the Level-C/OS partition and the per-core Level-A/B partitions) is viewed as a variable, which is determined by solving an mixed integer linear program (MILP) [24]. This MILP minimizes the task set's Level-C utilization while ensuring schedulability at all criticality levels. It is invoked only after an assignment of Level-A and -B tasks to cores has been obtained via bin-packing heuristics (*i.e.*, the MILP does not determine such an assignment). We will consider this MILP in greater detail later in Sec. 3 as a precursor to explaining how LLC and DRAM allocations can be determined for multi-mode systems.

The MC^2 implementation just described does not provide management for L1 caches, translation lookaside buffers, memory controllers, memory buses, or cache-related registers that can be a source of contention [35]. However, we assume PETs are determined via measurement, so such resources are implicitly considered when PETs are determined. We adopt a measurement-based approach because work on static timing analysis tools for multicore machines has not matured to the point of being directly applicable. Moreover, PETs are often determined via measurement in practice.

In recent work, we proposed extensions to MC^2 that permit tasks to share memory pages to support producer/consumer buffers [9] and to enable the usage of shared libraries [23]. Our mode-change extensions can be applied alongside these other extensions, but we do not consider that possibility here, for ease of exposition.[7]

[7]Delving into sharing would require providing further background and would complicate our experimental framework. We lack space for either.

Problem considered in this paper: supporting multiple modes. Our objective in this paper is to adapt the hardware-isolation mechanisms of MC^2 so that multiple functional modes can be supported. Each *mode* is defined by a set of periodic/sporadic tasks, with each such task set defined exactly as discussed at the beginning of Sec. 2. At any point in time, the system is either executing in a distinct mode or undergoing a transition from one mode to another. These transitions are enacted by a *mode-change protocol*.

Tasks from different modes do not execute at the same time (except perhaps briefly when a mode change is underway). However, memory pages for all tasks from all modes must be allocated in DRAM, unless secondary storage devices such as a solid-state disks are employed. While such devices are worthy of scrutiny, we do not consider them in this paper because their usage can cause relatively long mode-change latencies, which may be unacceptable in safety-critical domains. As seen above, DRAM allocations impact LLC allocations. Therefore, the main challenge we must address is to determine how to allocate tasks from all modes in both DRAM and the LLC so that schedulability is ensured for all modes. We address this challenge in Sec. 3 below. Additionally, we must add a mode-change protocol to MC^2's runtime scheduler and show that it gives rise to reasonable mode-change latencies. Due to space constraints, we defer consideration of these issues to Appendices A and B.

3 LLC/DRAM Allocation Problem

In this section, we give a more detailed overview of the MILP techniques mentioned in Sec. 2, and present our extensions of them to enable multiple modes.

3.1 Prior MILP Techniques

The MILP from prior work determines the LLC allocations shown in Fig. 3, assuming DRAM allocations are as shown at the bottom of the figure. This MILP, which is described at length in [24], is too complex to describe fully here. Instead, we opt to consider a simpler allocation problem that is sufficient for explaining the main ideas.

A simple motivating example. Consider a single task with a $4ms$ period running on a uniprocessor platform with a 4-way LLC. For this (very) simple task system, we explain how to construct a MILP that determines the number of ways to allocate to the task so that its resulting PET ensures system schedulability. The obvious choice would be to simply allocate all ways to the lone task. However, demonstrating the construction of a MILP for this example is still useful for understanding the MILP techniques we build upon.

Fig. 4 visually depicts the MILP constraints for this problem. The first set of constraints is determined based on PET measurement data. The figure shows five PET data points, one for each possible way allocation, with each point determined via measurement data. By introducing lines between adjacent data points, and constraining a solution to be above these lines, any PET value determined by the MILP will be in accordance with measurement data.

We must also ensure that the system is schedulable. Here,

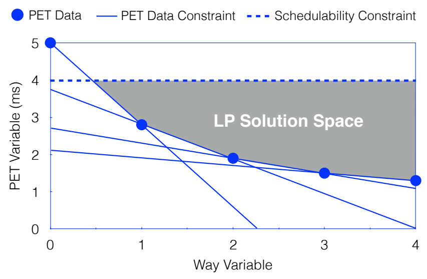


Figure 4: Illustration of MILP constraints.

with a single task, schedulability can be ensured by specifying just one constraint: the task's PET cannot be greater than its period, else the system will be over-utilized. This constraint is depicted as a dashed line in Fig. 4. The potential solution space, which is also depicted in the figure, is obtained by intersecting the half-spaces defined by the specified linear constraints. One could specify an objective function that would favor certain solutions within this space, but given the very simple nature of this motivating example, we will not bother to delve into objective functions just yet.

The actual allocation problem as a MILP. In the example above, schedulability was ensured by requiring the lone task's utilization to be at most 1.0. In MC^2, schedulability is similarly determined by checking a set of utilization-based constraints. Thus, the full MILP can be viewed as an extension of the simple example above in which linear constraints must be specified for many tasks while accounting for several utilization-based schedulability conditions. The full MILP also includes constraints on LLC allocation variables that ensure that certain LLC areas do not overlap. Furthermore, an objective function is added to minimize Level-C system utilization, as this will likely reduce tardiness at Level C.

In the simple example above, overheads affecting schedulability (*e.g.*, scheduling costs, context-switching times, *etc.*) were ignored. In the full MILP, such overheads are factored into the various linear constraints using standard overhead-accounting techniques that involve inflating PETs. Like PET data, these various overheads can be determined via measurement. Specific overhead values are treated as constants.

As described here, our prior MILP techniques determine *way allocations*, as shown in Fig. 3. Similar techniques can be applied in alternative allocation frameworks where LLC areas are sized by deducing *color allocations*, with way allocations being fixed. While it would be desirable to size LLC areas by deducing *both* color *and* way allocations, this is difficult (if not impossible) to do with only linear constraints. In particular, an LLC area's size is given by multiplying the number of ways and colors allocated to it. Thus, PETs become a nonlinear function of the number of allocated ways and colors when both of these parameters are viewed as variables. To the best of our knowledge, the only way to eliminate this nonlinear dependency is by requiring either way allocations or color allocations to be fixed.

Handling DRAM capacity constraints. In most work on real-time schedulability analysis, memory capacity is viewed

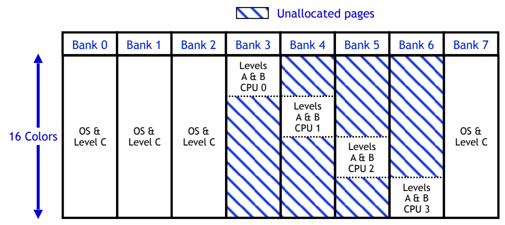


Figure 5: A closer look at the DRAM allocations in Fig. 3.

as an unconstrained resource. In reality, however, memory capacity certainly *is* limited. In recent work, we introduced DRAM-capacity constraints to our MILP-based optimization framework that ensure that the supply of pages within each DRAM bank is not over-allocated [23]. The introduction of these new constraints exposes a liability associated with the allocation scheme illustrated in Fig. 3. In particular, each DRAM bank has 16 page colors available, but within each Level-A/B bank, only four colors are used. Thus, 75% of the available space in these banks is unused, as shown in Fig. 5.

3.2 Allocating Multi-Mode Systems

We now propose several approaches for extending our prior MILP-based allocation scheme to account for the requirements of multi-mode systems. Clearly, a myriad of approaches could be devised for allocating LLC and DRAM space under MC^2 even without multiple modes being present. The introduction of modes creates even more possibilities, so it is not possible to consider every possible allocation approach. The approaches presented here are meant to be representative of the kinds of techniques that could be applied and were selected for inclusion because they expose interesting resource-allocation issues and tradeoffs. As we shall see, the presence of *shared tasks*—that is, tasks included in multiple modes—creates certain challenges. Thus, we initially assume such tasks are not present and then later address the challenges they introduce.

Multi-mode systems without shared tasks. The existing MC^2 framework could be used in a multi-mode system, but the tasks comprising all modes would have to be viewed as a single task system. Moreover, 75% of the available Level-A/B DRAM space would be wasted, as seen in Fig. 5. Our intent in devising other schemes is to reclaim this wasted space and use it to support tasks from different modes. We consider two schemes for doing this: *color-based allocation* (*CBA*) and *way-based allocation* (*WBA*).

Under CBA, each mode is assigned a set of colors in each Level-A/B DRAM bank such that these sets do not overlap. (The pages for the Level-A/B tasks in that mode are allocated from these assigned colors.) This is done by simply partitioning the 16 colors available across all banks into four disjoint groups, as shown in Fig. 6(a). With color groups so defined, assigning colors to modes is straightforward: denoting the color groups as Groups 0 through 3, and the modes as Mode 0, Mode 1, and so on, Mode i is assigned the Groups i *mod* 4, $i+1$ *mod* 4, $i+2$ *mod* 4, and $i+3$ *mod* 4 on DRAM banks 3, 4, 5, and 6, respectively (these are the Level-A/B DRAM banks, as seen in Fig. 3). CPU i is assigned Group i. This assignment is illustrated for Mode 1 in Fig. 6(a) with a gray shading. While this figure shows only four modes being assigned, we can keep assigning modes in this fashion as long as sufficient DRAM capacity is available.

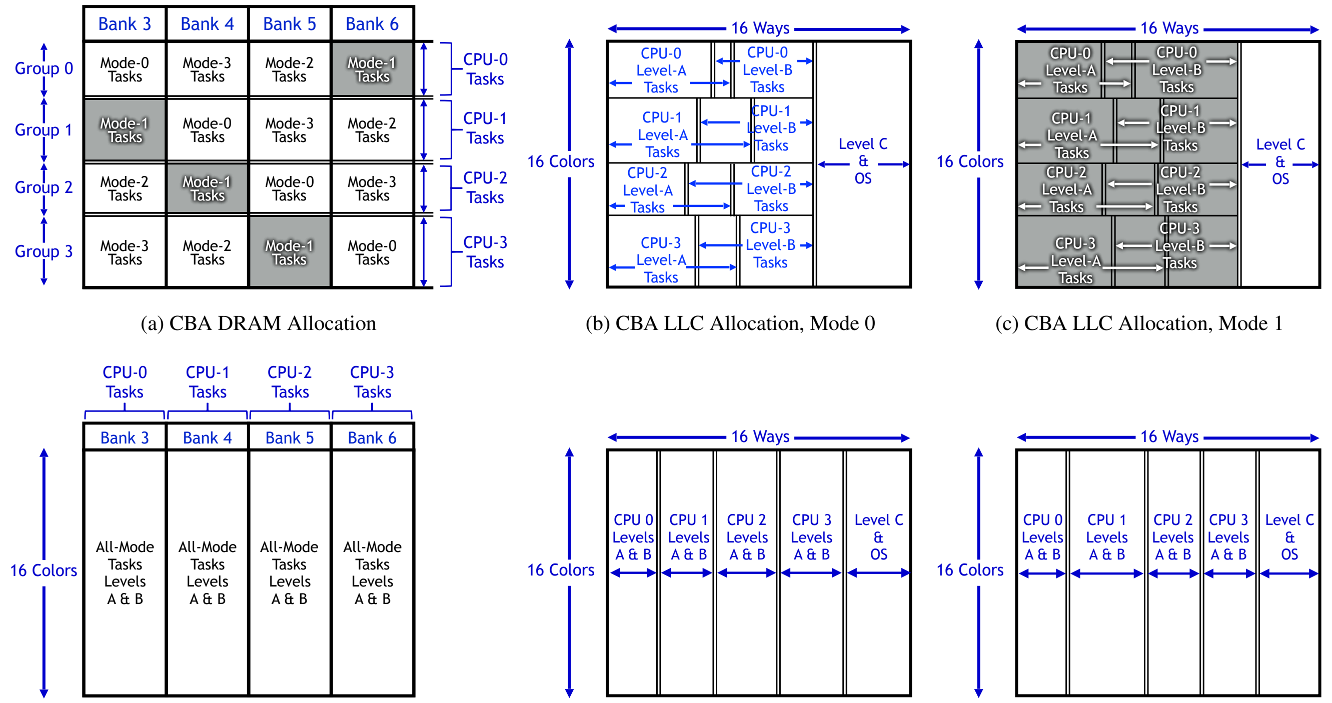
The four color groups need not have the same size. To determine the number of colors per group, we first assign to each group the minimum number of colors required to load all tasks into their assigned banks and groups; we then distribute any remaining unallocated colors to groups as evenly as possible. After determining these color assignments, way allocations in the LLC can be determined via the same MILP as before, as illustrated in insets (b) and (c) of Fig. 6. As seen in these insets, these allocations may be different for different modes. To better see the correspondence between the LLC and DRAM allocations in Fig. 6, we have shaded the allocations for Mode 1 in inset (c) as we did in inset (a).

As seen above, DRAM color allocations in CBA are determined before LLC allocations are optimized. Alternatively, we can extend our MILP techniques to determine DRAM and LLC allocations simultaneously, but this more exact approach is more costly and is difficult to apply at the scale of our schedulability study in Sec. 4. Still, this is an option worth considering, so we elaborate on it further in Appendix C.

Under WBA, the other allocation scheme considered here, each Level-A/B LLC area consists of all colors and a designated number of ways, as illustrated in insets (e) and (f) of Fig. 6. These LLC areas are all disjoint from each other and also from the Level-C/OS LLC area. Because each Level-A/B LLC area consists of all 16 colors, each Level-A/B DRAM bank can be fully utilized, as illustrated in Fig. 6(d). Each of these banks would be used to allocate pages to tasks from all modes. Under WBA, the MILP determines per-mode LLC way allocations. Like CBA LLC allocations, WBA LLC allocations may be different for different modes.

The main disadvantage of WBA compared to CBA is that, under WBA, fewer ways are provided per LLC area. This reduction in ways may be compensated for by an increase in colors, but our prior execution-time measurements for benchmark programs [24] suggests that some tasks are more sensitive to restrictions on ways and others to restrictions on colors. When considering multiple modes, each comprised of many tasks with different characteristics, it is difficult to say which scheme is best. To shed light on this issue, our schedulability study in Sec. 4 compares WBA, CBA, and other options. In this study, a general model of PETs based on measurement data was employed that reflects PET sensitivities to way and color allocations.

Multi-mode systems with shared tasks. We now consider problems that arise when tasks can be shared across modes. Such shared tasks are problematic because their presence implies that different modes must now share DRAM allocations. Note that this is only a problem when tasks are shared at Levels A or B, as all Level-C tasks share a common LLC area and each such task can be assigned pages from any Level-C/OS DRAM bank. We present two techniques for handling shared tasks, each of which can be applied together



(a) CBA DRAM Allocation (b) CBA LLC Allocation, Mode 0 (c) CBA LLC Allocation, Mode 1

(d) WBA DRAM Allocation (e) WBA LLC Allocation, Mode 0 (f) WBA LLC Allocation, Mode 1

Figure 6: DRAM and LLC allocations under CBA (top row of figures) and WBA (bottom row of figures).

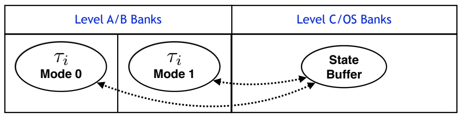


Figure 7: DRAM allocation for a task replicated across two modes.

with either CBA or WBA.

The first technique is to *partition shared tasks* (PST). Under PST, all Level-A/B shared tasks are assigned to CPU 0 and are allocated DRAM pages from the Level-A/B bank associated with CPU 0. Non-shared Level-A/B tasks are assigned to the other CPUs. Under CBA, shared tasks are color-partitioned from other tasks in the LLC, and under WBA, they are way-partitioned. Modifying the MILP for either CBA or WBA to apply PST is straightforward.

The second technique is to *replicate shared tasks* (RST). Under RST, each shared task is replicated for each mode in which it runs, and each replica is treated as a non-shared task. This technique is relatively straightforward to apply, but one complication does arise. If a replicated task needs to preserve state information across its jobs, then that information must be retained in memory, and this creates data-sharing relationships across modes that can break MC^2's hardware isolation guarantees. Fortunately, in our prior work on supporting producer/consumer shared data buffers in MC^2 [9], we proposed several approaches that can ease such problems.

Consider, for example, Fig. 7, which shows a Level-A/B task τ_i that has been replicated between Modes 0 and 1, where each replica is assigned to a different DRAM bank. If state information must be retained across the jobs of τ_i, then that information can be stored in a buffer. Under the approach from [9] that is most directly applicable to the situation here, the buffer would be allocated in the Level-C/OS DRAM banks, as shown in Fig. 7, and configured to bypass the LLC, so that LLC isolation is not compromised. Since Level C is provided no cross-core DRAM isolation, accesses to the state buffer by jobs of τ_i do not affect Level-C isolation properties. However, accesses to the buffer by jobs of task τ_i can experience interference from Level-C tasks and the OS, which was not possible before. This source of interference would have to be taken into account in the measurement process for determining PETs.

Note that PST and RST can both be applied in the same system, with the choice being made on a per-task basis. However, to keep the experimental study in Sec. 4 at a manageable level, we do not examine this possibility further.

Supporting task criticality-level changes. As noted in Sec. 1, it may be desirable to allow a task to undergo a criticality change when a mode change occurs. Such a task can be supported by creating replicas of it as needed at different criticality levels and in different modes. RST provides the necessary functionality to support such replicas.

4 Evaluation

Our extensions to MC^2 to support mode changes involved altering both its offline allocation component, as described in the prior section, and also its runtime scheduler. We evaluated the former by performing a large-scale schedulability study

Mult-Mode Task Management	PST	RST
NAIVE	N/PST	N/RST
CBA	CBA/PST	CBA/RST
WBA	WBA/PST	WBA/RST

Table 1: Considered allocation variants.

and the latter via case-study experiments in which mode-change latencies were measured. Due to space constraints, we discuss our modifications to MC^2's runtime component in Appendix A and our case-study experiments in Appendix B. In this section, we discuss our schedulability study. The code for our study is available online.[8]

Schedulability study overview. We assessed the efficacy of the allocation schemes presented in Sec. 3.2 by evaluating the schedulability of randomly generated task systems under the MC^2 variants listed in Tbl. 1, as well as the HRT uniprocessor earliest-deadline-first scheduler, denoted U-EDF, with all banks allocated to all modes.[9] The latter reflects current industry practice for eliminating shared-hardware interference by simply disabling all but one core.

The CBA and WBA variants in Tbl. 1 use the associated allocation methods described in Sec. 3.2. Under the NAIVE variants, if no shared tasks exist, DRAM is allocated as in Fig. 5, assuming that all tasks from all modes comprise one all-encompassing task system. However, way allocations in the LLC are determined on a per-mode basis as illustrated in Fig. 3 by solving a MILP for each mode (way allocations may be different for different modes). Shared tasks can be easily introduced under this allocation scheme by using the PST and RST approaches discussed earlier in Sec. 3.2. In addition to the MC^2 variants listed in Tbl. 1, we also consider the ALL variant, which simply involves checking whether any of the variants in Tbl. 1 produces a schedulable allocation. This variant is interesting to consider because it shows the potential value of trying multiple allocation approaches.

We used the following limits on available DRAM on the Cortex A9 platform when assessing schedulability. Each bank allocated to Levels A and B has approximately 32,000 pages available for task allocation and approximately 2,000 pages of each color. Approximately 73,000 pages are available to Level C after accounting for LITMUS$^{\mathrm{RT}}$ OS allocations. All tasks were assumed to statically link to libraries and not dynamically allocate memory, so the only DRAM consumption to consider beyond that required by the OS was static task page allocation and shared state buffers.

Task-system generation. We extended an evaluation framework used extensively by us in prior work [9, 11, 23, 24, 25] to randomly generate task systems while accounting for DRAM consumption and multiple modes. Under this framework, PETs are determined at Level B (resp., Level C) based on measured worst-case (resp., average-case) execution-time data for benchmark tasks, as discussed in detail in prior

Category	Choice	Level A	Level B	Level C
1: Mode Count	Few	{2, 3, 4, 5, 6, 7}		
	Many	**{8, 9, 10, 11, 12}**		
2: Criticality Utilization Percent	C-Light	[29, 56)	[29, 56)	[10, 25)
	C-Heavy	[9, 33)	[9, 33)	[45, 78)
	All-Mod.	**[28, 39)**	**[28, 39)**	**[28, 39)**
3: Period (ms)	Short	{3, 6}	{6, 12}	[3, 33)
	Medium	**{12, 24}**	**{24, 48}**	**[12, 100)**
	Long	{48, 96}	{96, 192}	[50, 500)
4: Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Medium	[0.02, 0.1)	[0.05, 0.2)	[0.1, 0.4)
	Heavy	**[0.1, 0.3)**	**[0.2, 0.4)**	**[0.4, 0.6)**
5: Page Count in Hundreds	Light	[1.5, 3)	[1.5, 3)	[2, 7):0.75 [6, 13):0.25
	Medium	**[3, 5)**	**[3, 5)**	**[4, 9):0.75** **[10, 30):0.25**
	Heavy	[5, 7)	[5, 7)	[6, 11):0.75 [13, 70):0.25
6: Shared Utilization	Light	[0%, 20%)	[0%, 20%)	[0%, 20%)
	Heavy	**[50%, 70%)**	**[50%, 70%)**	**[50%, 70%)**
7: Criticality Change		**20%**		
8: Max Reload Time	**Light**	**[0.01, 0.1)**	**[0.01, 0.1)**	**[0.01, 0.1)**
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
9: State		**[0%, 10%)**		
10: Color Sensitivity	**Reduced**	**[70%, 90%)**		
	Regular	0%		

Table 2: Task-set parameters and distributions. In Category 5, last column, $I{:}P$ denotes that interval I is selected with probability P.

work [24]. Similarly to Fig. 4, these PETs are a function of ways and colors. Level-A PETs are obtained by applying a 50% inflation factor to Level-B PETs. For replicated shared tasks, these PETs must be adjusted to reflect state buffer usage, and this can be done by applying buffer-accounting methods presented by us previously [9].

To generate task and task-set parameters, ten distributions must be selected from the categories listed in Tbl. 2. Using the selected distributions, multi-mode task systems are then generated for each considered allocation variant by following a step-wise process that is explained in detail in our prior work [24] and refined here to be applicable to multi-mode systems. We provide below a high-level overview of our refinements by considering each step in turn, assuming the distributions highlighted in bold in Tbl. 2 have been selected.

Step 1: Determine the number of modes by using the distribution selected in Category 1. The highlighted selection indicates that the number of modes will be selected at random from the range $\{8, 9, \ldots, 12\}$.

Step 2: Generate the first mode. Generally, this is done by generating a task set for the U-EDF variant and then modifying that task set for the other considered allocation variants by adjusting PETs to reflect differences in LLC-allocation and hardware-isolation choices. Such modifications are driven by measurement data taken on the Cortex A9 following an approach described in detail in our prior work [24].[10] The highlighted selection in Category 2 indi-

[8]https://wiki.litmus-rt.org/litmus/Publications

[9]Bank contention is not possible when only one core runs. Hence, it is safe to allocate any bank to any task under U-EDF.

[10]Note that generating one mode simply requires generating a single task set, so our prior work can be directly applied here.

cates that the percentage of tasks at each criticality level will be in the range $[28, 39)\%$ (totaling to 100%). The highlighted selections in Categories 3–5 specify how per-task U-EDF parameters are determined. For example, each Level-A task will have a period of either $12ms$ or $24ms$, a U-EDF utilization in the range $[0.1, 0.3)$, and require 300 to 499 pages in DRAM. After a task set has been fully defined for the U-EDF variant, that task set is adjusted to create a variant for each allocation scheme, as already mentioned.

Step 3: Generate all other modes. Subsequent modes are generated following a similar process as outlined in Step 2, except that shared tasks have to be determined. This is done using the distribution selected in Category 6. The highlighted selection in this category indicates that, at each criticality level, the number of tasks shared between a newly generated mode and the prior mode is such that these tasks have a combined U-EDF utilization of $[50, 70)\%$ of the prior mode's U-EDF utilization at that level. Ordinarily, any task retained from the prior mode retains the same criticality level. However, the distribution in Category 7 is applied to designate that some shared tasks undergo a criticality change. In particular, of the tasks that are shared with Level C of the new mode, 20% are taken from Level B of the prior mode.[11]

Step 4: Add to all modes special per-core Level-A mode-change-handling tasks as used in the mode-change protocol described in Appendix A. Each such task has a period equal to the shortest system-wide Level-A period, and Level-A, -B, and -C PETs of 254μs, 169μs, and 83μs, respectively. These PETs were determined from measured execution times for an actual mode-change-handling task on the Cortex A9.

Step 5: Adjust generated PETs to account for implementation-related overheads, shared-buffer copy times, and differences in PET sensitivy to LLC-way and -color allocations. These adjustments are affected by the distributions selected for Categories 8, 9, and 10. For example, the highlighted distributions from Category 8 indicate that the maximum LLC reload time under U-EDF for any task after a preemption or migration is $[1, 10)\%$ of its U-EDF PET, that from Category 9 indicates that $[0, 10)\%$ of each task's pages must be stored in a state buffer (if replicated), and that from Category 10 indicates that the variation of a task's PETs with respect to allocated LLC colors is reduced by $[70, 90)\%$ (indicating lesser sensitivity to color allocations). The adjustments made in this step are based on measurement data obtained on the Cortex A9.

Step 6: For each MC^2 variant, assign Level-A and -B tasks to cores using the worst-fit decreasing heuristic discussed in [24], with obvious modifications for the PST variants to ensure that all shared tasks are assigned to Core 0. Under the RST variants, each replica of a Level-A or -B shared task is simply treated as an independent task.

Step 7: Test the schedulability of the resulting multi-mode task system under each evaluated allocation variant.

The distributions in Tbl. 2 were defined to enable the systematic study of different factors impacting schedulability, such as MC analysis, DRAM constraints, and mode relationships, and were selected in a way to strike a balance between having a manageable study and covering a wide range of choices. Additionally, much of the task-system generation process is based on actual measurement data.

We denote each combination of distribution choices using a tuple notation. For example, (Many, All-Mod., Medium, Heavy, Medium, Heavy, Light, Reduced) denotes using the Many, All-Mod., Medium, *etc.*, distribution choices for those categories in Tbl. 2 that have multiple options. We call such a combination a *scenario*. We considered all possible such scenarios, and for each task-system utilization in each scenario, we generated enough task systems to estimate mean schedulability to within ± 0.05 with 95% confidence with at least 100 and at most 300 task systems.

Schedulability results. In total, we evaluated the schedulability of over 4 million randomly generated task systems, which took roughly 190 CPU-days of computation. (The MILP typically required less than a second per considered task system to execute.) From this abundance of data, we generated 1,296 schedulability plots, of which three representative plots are shown in Fig. 8. The full set of plots is available online [10].

Each schedulability plot corresponds to a single scenario. To understand how to interpret these plots, consider Fig. 8(a). In this plot, the x-axis represents the nominal[12] per-mode U-EDF utilization. The actual U-EDF utilization varies by up to 50% from mode to mode to reflect variations expected in real-world task systems. In this plot, the circled point indicates that 43% of the generated task systems with a nominal U-EDF utilization of 3.0 were schedulable under the N/PST variant. Note that, because the x-axis represents system utilizations under the single-core HRT U-EDF variant, it is possible under MC^2 to support systems with a nominal U-EDF utilization exceeding four, as MC provisioning and hardware management decrease PETs.

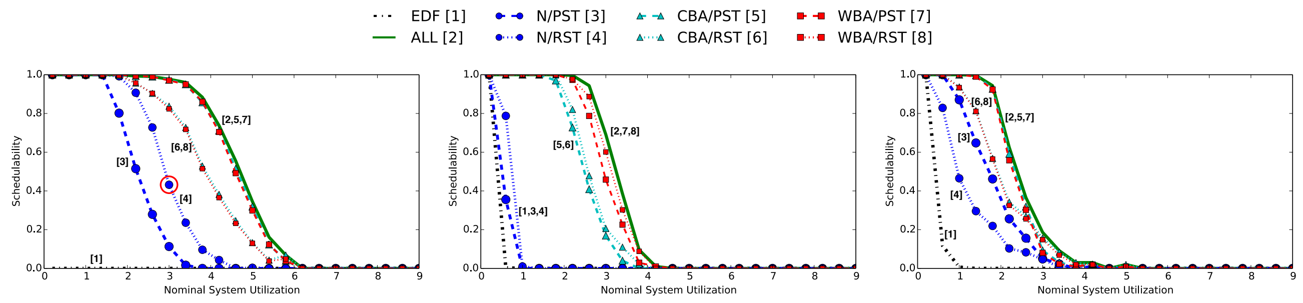
We now state several observations that follow from the full set of collected schedulability data. We illustrate these observations using the data presented in Fig. 8.

Obs. 1. [Naive vs. other] When comparing the PST variants, schedulability under CBA and WBA was on average 367% and 477% better, respectively, than schedulability under NAIVE. Respective percentages for the RST variants were 265% and 326%.

All insets of Fig. 8 show moderate to significant schedulability gains for the non-NAIVE variants over the NAIVE variants. These gains underscore *the importance of considering limited DRAM space in multi-mode systems*.

[11]It is likely in industry settings that tasks requiring Level-A certification will require Level-A certification in all modes. As a result, we assume no task changes criticality level to or from Level A.

[12]The term *nominal* has a technical definition that we omit due to space constraints. For a general understanding of our plots, it suffices to substitute "average" for "nominal."



(a) (Many, C-Heavy, L., H., M., L., H., Reg.) (b) (Many, All-Mod, L., M., M., L., L., Reg.) (c) (Few, All-Mod, L., M., H., H., L., Red.)

Figure 8: Representative schedulability plots.

Obs. 2. [PST vs. RST] Schedulability under N/PST, CBA/PST, and WBA/PST over all scenarios combined was 22%, 31%, and 28% better than that under N/RST, CBA/RST, and WBA/RST, respectively.

In insets (a) and (c) of Fig. 8, most of the PST variants outperform their RST counterparts (*e.g.*, CBA/PST outperforms CBA/RST). The RST variants are negatively affected by greater DRAM requirements than the PST variants for replicated pages.

Obs. 3. [CBA vs. WBA] Schedulability under WBA/PST and CBA/PST was comparable (less than a 0.02 difference in schedulable utilization) in scenarios with regular color sensitivity (refer to Category 10 in Tbl. 2). Similar results were seen in reduced-color-sensitivity scenarios.

In comparing the scenarios in insets (a) and (c) of Fig. 8, we see little difference in WBA/PST vs. CBA/PST, even though each of these scenarios exhibits different sensitivities of PETs to available colors. Schedulability under the two respective RST schemes is also similar in these two insets. Despite disadvantages for WBA allocations in color-sensitivity, WBA and CBA schedulability were comparable nonetheless.

Obs. 4. Of all considered schemes, WBA/PST and CBA/PST performed the best, achieving on average over 94% of the schedulability of ALL.

This observation is supported by all insets of Fig. 8.

Given the nature of our study, these observations naturally hinge on our experimental setup. However, we have taken care to ensure that a wide range of system configurations were considered.

5 Prior Related Work

This work follows a long line of research examining shared-resource contention in real-time systems [26]. Prior efforts have focused on issues such as cache partitioning [3, 17, 21, 39, 38], DRAM controllers [4, 13, 18, 19, 27], and bus-access control [1, 2, 12, 14, 15, 32]. Other work has focused on reducing shared-resource interference when per-core scratchpad memories are used [34], accurately predicting DRAM access delays [20], throttling lower-criticality tasks' memory accesses [41], allocating memory [40], and enhancing temporal isolation by managing shared pages [9, 22, 23]. In evaluating one recently proposed cache-partitioning scheme, vCAT [39], a dual-mode use case was considered. However, that use case was quite simplistic: in addition to having only two modes, all tasks were shared and no DRAM constraints were considered.

To our knowledge, we are the first to consider in detail complexities that arise when attempting to support multiple modes while ensuring hardware isolation under the notion of MC scheduling espoused by Vestal [36], which was proposed with the express intent of *achieving better platform utilization*. Several of the aforementioned papers do target MC systems [4, 12, 14, 15, 18, 19, 27, 31, 41], but only peripherally touch on the issue of achieving better platform utilization, if at all. Hardware isolation under Vestal's notion of MC scheduling has been considered in several prior MC^2-related papers by our group [9, 16, 23, 24, 29, 37], but these papers do not consider multi-mode systems.

Real-time mode-change protocols are a well-studied topic, but most of the classic work on this topic focuses on uniprocessors. A survey of such work with a fairly comprehensive bibliography has been produced by Real and Crespo [33]. In some work pertaining to Vestal's notion of MC schedulability analysis, a task exceeding a PET can cause a *criticality mode change* in which lower-criticality tasks may be dropped. Such mode changes are quite different from functional mode changes as considered in this paper. Burns recently published a survey paper in which the differences between these two kinds of mode changes are discussed at length [5].

6 Conclusion

In this paper, we have provided extensions to MC^2 for supporting multiple functional modes. As we have seen, tasks shared across multiple modes pose a particular challenge, because they cause hardware-allocation decisions affecting different modes to become intertwined. Our extended MC^2 framework not only supports such tasks but also allows them to change their criticality levels.

When supporting mode changes in a framework like MC^2, where both hardware-isolation properties must be ensured and MC analysis assumptions are employed, numerous resource-allocation tradeoffs exist. The various offline allocation approaches studied in this paper were selected as reasonable candidate solutions that expose interesting tradeoffs. We evaluated these tradeoffs via a large-scale overheadaware schedulability study. In this study, the WBA and CBA schemes tended to provide significant schedulability gains over the NAIVE schemes, and the WBA/PST and CBA/PST variants faired the best overall. When considering multi-mode systems, mode-change latencies are also a concern. In experiments described in Appendix B, latencies for our mode-change protocol tended to be quite reasonable, usually on the order of $200ms$ or less.

The results of this paper open up many avenues for future work. For example, a wider range of hardware-allocation options could be explored than was possible to cover in the space available. Additionally, as explained in Appendix A, several implementation options for supporting mode-change protocols exist that warrant further attention. Finally, we have assumed in this paper that all tasks from all modes fit within DRAM. This is a very desirable property, but some systems may exist in practice in which DRAM alone is insufficient and thus secondary storage devices such as solid-state disks must be used. Understanding how to page tasks to and from disks in a way that is reflective of criticality concerns is an interesting topic that warrants further study.

References

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

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

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

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

[5] A. Burns. System mode changes - general and criticality-based. In *WMC '14*.

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

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

[8] Certification Authorities Software Team (CAST). Position paper CAST-32A: Multi-core processors, Nov. 2016.

[9] M. Chisholm, N. Kim, B. Ward, N. Otterness, J. Anderson, and F.D. Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *RTSS '16*.

[10] M. Chisholm, S. Tang, N. Kim, N. Otterness, J. Anderson, D. Porter, and F.D. Smith. Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems. Full version of this paper, available at `http://jamesanderson.web.unc.edu/papers/`.

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

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

[13] D. Guo and R. Pellizzoni. A requests bundling DRAM controller for mixed-criticality systems. In *RTAS '17*.

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

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

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

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

[18] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and P. Cazorla. A dual-criticality memory controller (DCmc) proposal and evaluation of a space case study. In *RTSS '14*.

[19] H. Kim, D. Broman, E. Lee, M. Zimmer, A. Shrivastava, and J. Oh. A predictable and command-level priority-based DRAM controller for mixed-criticality systems. In *RTAS '15*.

[20] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS '14*.

[21] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical OS-level cache management in multi-core real-time systems. In *ECRTS '13*.

[22] H. Kim and R. Rajkumar. Memory reservation and shared page management for real-time systems. *Journal of Sys. Arch.*, 60:165–178, Feb. 2014.

[23] N. Kim, M. Chisholm, N. Otterness, J. Anderson, and F.D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *RTAS '17*.

[24] N. Kim, B. Ward, M. Chisholm, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Sys.*, 2017 (to appear).

[25] N. Kim, B. Ward, M. Chisholm, C.-Y. Fu, J. Anderson, and F.D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *RTAS '16*.

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

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

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

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

[30] J. Musmanno. Data intensive systems (DIS) benchmark performance summary, Aug. 2003.

[31] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity environment. In *ECRTS '14*.

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

[33] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Sys.*, 26(2):161–197, 2004.

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

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

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

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

[38] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee. Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation. In *RTAS '16*.

[39] M. Xu, L. T. X. Phan, H. Y. Choi, and I. Lee. vCAT: Dynamic cache management using CAT virtualization. In *RTAS '17*.

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

[41] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memory access control in multiprocessor for real-time systems with mixed criticality. In *ECRTS '12*.

A Mode-Change Protocol Implementation

In Sec. 3, we focused on hardware-allocation options. Here, we turn our attention to changes we made to MC^2's runtime scheduler. The code for these changes is available online.[13]

Protocol requirements. A mode-change protocol is invoked when a *mode-change request* (*MCR*) is initiated by the application. When an MCR occurs, the mode-change protocol *enacts* the requested mode change. The goal is to do so as quickly as possible while maintaining safety (*e.g.*, terminating a partially completed job from the old mode might be problematic as an inconsistent system state may result).

In designing our mode-change protocol, we sought to maintain as an invariant that, when changing from one mode to another, the jobs of high-criticality Level-A/B tasks from both modes do not experience any performance degradation or interference. Since such tasks are HRT and periodic and have harmonic periods (and start execution at time 0), the most straightforward response is to enact mode changes only at Level-A/B hyperperiod boundaries, because any Level-A/B job from the prior mode will have finished by then.

Unfortunately, this simple approach could be problematic for Level-C tasks. Each such task is sporadic and SRT, so one from the old mode could have a partially completed job at any Level-A/B hyperperiod boundary. Such a job can be either run to completion or aborted. However, in the latter case, some undo code may be needed in order to avoid introducing inconsistencies. These approaches are subject to many tradeoffs, the resolution of which would be tightly coupled with application-level requirements. Such requirements are beyond the scope of this paper, so we take a simple approach here: after an MCR is received, we first halt any Level-C task that is part of the old mode but not the new mode and wait until any currently executing jobs of such halted tasks complete; we then enact the mode change at the first following time point that is a hyperperiod boundary with respect to all Level-A/B tasks on all CPUs.

High-level implementation description. Implementing our mode-change protocol involved introducing new state information into the MC^2 scheduler. Fig. 9 shows these states. The system is in the *Stable* state when the MC^2 scheduler is scheduling tasks from a single mode and no mode change has been requested. When a mode change is requested via an MCR, the system transits to the *Pending* state, where it remains until all partially completed jobs of halted Level-C tasks have completed, as discussed above. When the last such job completes, the system transits to the *Ready* state, where it remains until the next Level-A/B hyperperiod boundary of the current mode is reached, at which point the mode change is enacted and the system transits back to the Stable state.

In our implementation, any task can issue an MCR via a system call in which a new mode is specified. While in the Pending state, any new MCRs that are received are ignored. In LITMUS$^{\mathrm{RT}}$, the Pending state is implemented using a scheduling plugin that causes a callback in the scheduler to

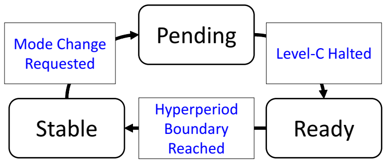


Figure 9: State machine of mode change process.

be invoked whenever real-time jobs complete. This callback is used to keep track of the number of Level-C jobs that are still executing. The LITMUS$^{\mathrm{RT}}$ plugin also makes it possible to prevent the release of new jobs of halted Level-C tasks.

The Ready-to-Stable transition is actually enacted by per-CPU Level-A mode-change-handling tasks that exist in every mode. These special tasks' periods are equal to the Level-A/B hyperperiod in the current mode, and they are given the highest priorities within each mode,[14]guaranteeing that they run first after the hyperperiod boundary. For each mode, the system maintains a global list of Level-C tasks along with core-specific lists of Level-A and -B tasks. When the system is in the Ready state, the special tasks provide the scheduler with the correct lists of tasks for the new mode.

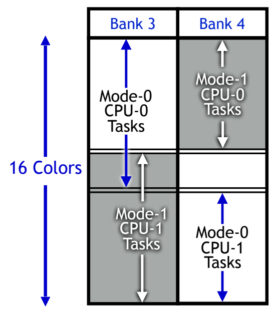
B Case Studies

For the mode-change protocol proposed in Appendix A to be considered reasonable, mode-change latencies should not be too high. To get a sense of such latencies, we conducted case-study experiments in which various multi-mode task systems were executed and mode-change latencies recorded. We constructed ten such task systems for each scenario shown in Fig. 8 using synthetic benchmark programs as exemplars of more-deterministic computations and programs from the DIS benchmark suite [30] as exemplars of less-deterministic computations. We ran each of these task systems for five minutes on an ARM Cortex A9 platform. We collected 3.5GB of scheduling data and measured mode-change latencies by randomly initiating MCRs, with an average spacing between MCRs of approximately one second. For each initiated MCR, we measured the length of time between the following Level-A/B hyperperiod boundary and the Level-A/B hyperperiod boundary where the requested mode change was actually enacted (the two might be the same if Level-C tasks have been halted by the first Level-A/B hyperperiod boundary after the MCR is received). The time between the occurrence of an MCR and the next Level-A/B hyperperiod boundary is uninteresting because the requested mode change cannot occur in this interval according to our protocol. The length of this interval is very short in any case, and if necessary, our reported mode-change latencies could be pessimistically rounded up by the length of a single Level-A/B hyperperiod.

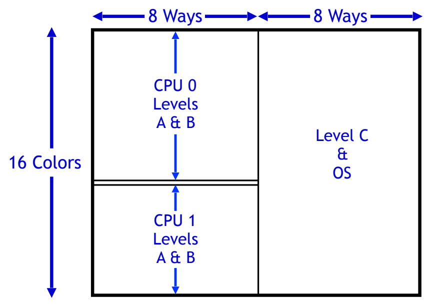
Across all of our experiments, the maximum latency was $193.70ms$, and the average latency was $60.52ms$. These latencies are in accordance with release delays of new-mode tasks given in [33].

[13]https://wiki.litmus-rt.org/litmus/Publications

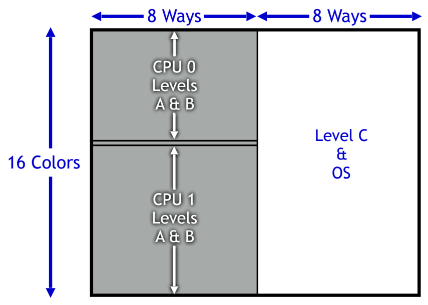
[14]In our schedulability study presented in Sec. 4, mode-change-handling tasks were given the *shortest* period at Level A because MC^2 schedulability analysis requires the highest-priority Level-A task to have the shortest period. Allocating the shortest period in analysis conservatively models runtime behavior while meeting MC^2 schedulability-analysis requirements.



(a) LBA DRAM Allocation

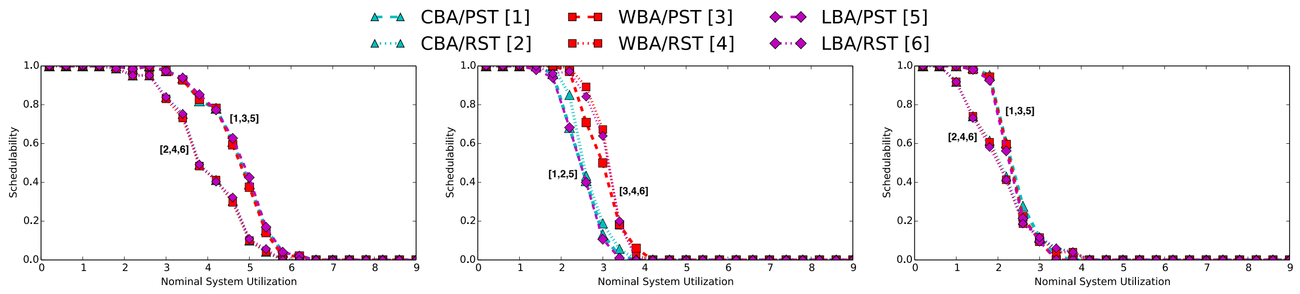


(b) LBA LLC Allocation, Mode 0



(c) LBA LLC Allocation, Mode 1

Figure 10: DRAM and LLC allocations under LBA.



(a) (Many, C-Heavy, L., H., M., L., H., Reg.) (b) (Many, All-Mod, L., M., M., L., L., Reg.) (c) (Few, All-Mod, L., M., H., H., L., Red.)

Figure 11: Representative schedulability plots for LBA compared to CBA and WBA.

C MILP-Based DRAM Allocation

In Sec. 3, we noted an alternative to CBA, denoted here as *MILP-based allocation* (LBA). Under LBA, our MILP techniques for determining LLC color allocations are extended to determine DRAM color allocations as well, with LLC way allocations being fixed. Here we briefly explain this allocation scheme and the changes to our MILP and compare the performance of LBA to CBA and WBA.

Consider a task set with two modes and all Level-A and -B tasks allocated to the first two CPUs, with no tasks shared between modes. Fig. 10 shows a simplified version of LBA for the LLC and DRAM banks assigned to these tasks. In this simplified version, LBA determines a placement of the doubled lines in each inset. Since the MILP determines an allocation of DRAM colors, it simultaneously determines an allocation of LLC colors. As a result, LLC ways must be fixed in the MILP. Note that, while the modes can share color allocations on the same bank, each mode maintains LLC-color isolation at Levels A and B, as demonstrated by the shaded Level-A and -B allocations for Mode 1 in Fig. 10. Also, although per-bank color allocations for each mode are (for simplicity) shown as contiguous in Fig. 10, this is not required. For instance, in Mode 0, tasks on CPU 0 could be allocated Colors 2, 5, and 11 in Bank 3, as long as these colors are not used by Mode-0 tasks on CPU 1 in Bank 4.

Modifying the MILP that determines LLC color allocations so that it also determines DRAM color allocations is relatively straightforward, although the new MILP is (obviously) more complex. However, since LBA fixes way allocations, it still may have disadvantages over CBA and WBA.

Overall, however, LBA explores a much larger combination of LLC and DRAM allocations than either CBA or WBA. CBA and WBA only search for schedulable LLC allocations under one DRAM allocation per task set. LBA's more thorough exploration of available hardware allocations comes at the expense of greater computational complexity to model DRAM page allocations. As a result, LBA demonstrates a tradeoff between achievable schedulability and runtime performance when compared to CBA and WBA.

We evaluated this tradeoff by testing the schedulability of LBA with PST and RST for a subset of scenarios in our larger schedulability study (specifically, the scenarios shown in Fig. 8) and compared it to schedulability under the corresponding CBA and WBA variants. To curb potential schedulability loss due to fixing way allocations, we considered LBA under three fixed way allocations: four, eight, and twelve ways allocated to Level C with remaining ways allocated to Levels A and B. The results of these experiments are shown in Fig. 11. Observe that CBA and WBA performed nearly as well as LBA under either RST and PST. These results indicate that our CBA and WBA heuristics perform well even without the more rigorous and computationally intensive search for schedulable DRAM and LLC allocations done by LBA.

While a wider range of experiments would be desirable for comparing the performance of CBA and WBA to that of LBA, LBA's runtime performance is a limiting factor affecting the number of experiments that can be performed in a reasonable amount of time. The experiments for Fig. 11 took 132 CPU-days to run.