Allowing Shared Libraries while Supporting Hardware Isolation in Multicore Real-Time Systems

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

Abstract—**The desire to support real-time applications on multicore platforms has led to intense recent interest in techniques for reducing memory-related hardware interference. These techniques typically rely on mechanisms that ensure per-task isolation properties with respect to cache and memory accesses. In most prior work on such techniques, any sharing of memory pages by different tasks is defined away, as sharing breaks isolation. In reality, however, sharing is common. In this paper, one source of sharing is considered, namely, the usage of shared libraries. Such sharing can be obviated by statically linking libraries, but this solution can degrade schedulability by exhausting memory capacity. An alternative approach is proposed herein that allows library pages to be shared while preserving isolation properties. This approach is presented in the context of the MC^2 framework and a schedulability-based evaluation of it is presented. Such an evaluation must necessarily consider memory-capacity limits. As a secondary contribution, this paper considers such limits for the first time in the context of MC^2.**

I. Introduction

The use of multicore platforms in safety-critical domains has been stymied by a problem that has been dubbed the "one-out-of-m" problem [8]: when certifying the real-time correctness of a system running on m cores, analysis pessimism can easily negate the processing capacity of the additional $m - 1$ cores. In effect, a system may be able to utilize only "one core's worth" of capacity even though m cores are available. This problem has led to the common practice in many settings of simply disabling all but one core.

The roots of this problem can be traced to two sources: undue task-provisioning pessimism, and excessive interference and unpredictability in accessing shared hardware. These issues are interrelated, and in resolving them, many tradeoffs exist. In ongoing work, our group has been exploring these tradeoffs in the context of a scheduling framework called MC^2 (mixed-criticality on multicore) [7, 8, 12, 20, 24, 31], which supports both mixed-criticality (MC) techniques for easing provisioning pessimism [30] and mechanisms for managing shared-hardware resources, particularly the shared last-level cache (LLC) and DRAM memory.

Work supported by NSF grants CPS 1239135, CNS 1409175, CPS 1446631, and CNS 1563845, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.

978-1-5090-5269-1/17/$31.00 ©2017 IEEE

These hardware-management mechanisms rely on techniques that ensure that tasks are isolated from one another with respect to LLC and memory accesses. In reality, however, tasks often share memory pages. This poses a problem because sharing breaks isolation. In recent work, we considered the implications of sharing as caused by the usage of shared data buffers [7]. In this paper, we consider another common source of sharing, the usage of shared libraries.

Shared libraries. To deal with shared libraries in our prior work, we assumed that they were always *statically linked*, meaning that all needed libraries were replicated on a per-task basis to eliminate any sharing. In the schedulability studies that we conducted to evaluate the effects of hardware management in MC^2, this solution came "for free" because we did not consider memory to be a constrained resource when checking schedulability. In practice, however, the combined memory footprint of all tasks and the operating system (OS) obviously must fit within the provided physical memory. Additionally, the available space might be further constrained for other reasons. For example, many safety-critical systems operate in different modes, so additional tasks might occupy memory beyond those currently running.

Unfortunately, when memory *is* considered as a constrained resource, the wasteful practice of fully replicating shared libraries can degrade schedulability significantly. Evidence of this can be seen in Fig. 1, which gives a portion of a plot that we consider only briefly here but in more detail later. This plot is taken from an experiment in which task systems were generated at random and their schedulability checked under MC^2. The curve labeled "NI-IDL" is illustrative of those given in our prior work: it was obtained without regard for memory limitations. The curve labeled "NI-STC" was similarly obtained, except that limits on memory capacity were accounted for in checking schedulability, with static linking assumed for shared libraries. As seen, the use of static linking significantly degraded schedulability.

Contributions. The primary contribution of this paper is to consider for the first time whether shared libraries can be supported in systems that ensure hardware isolation without inordinately degrading schedulability. We propose an approach for doing so in the context of MC^2 that is an intermediate between the extremes of replicating

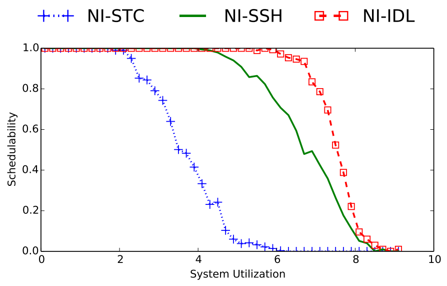


Fig. 1: An example schedulability plot. The manner in which such plots are interpreted will be made clear later. It suffices to know at this point that curves that degrade to zero further to the right indicate better schedulability.

libraries on a per-task basis and not replicating them at all. The effectiveness of this approach is reflected by the curve labeled "NI-SSH" in Fig. 1; this curve is similar to the others, except that the proposed approach has been applied. Our approach involves replicating shared libraries and allowing only certain groups of tasks to share a given replica.

To evaluate the proposed approach, we conducted a large-scale overhead-aware schedulability study. Any such study must necessarily consider memory-capacity limits when checking schedulability. As such, the work in this paper forced us to consider such limits for the first time in the context of MC^2. For technical reasons explained later, the introduction of such limits gives rise to new viable options for configuring LLC space and memory partitions in MC^2 that would have been pointless to consider before. These options have been included in our schedulability study.

Organization. In the following sections, we provide needed background (Sec. II), describe our library-sharing approach (Sec. III), consider the impacts of viewing memory as a constrained resource (Sec. IV), present the above-mentioned schedulability study (Sec. V), discuss related work (Sec. VI), and conclude (Sec. VII).

II. Background

We begin by reviewing needed background material.

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

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

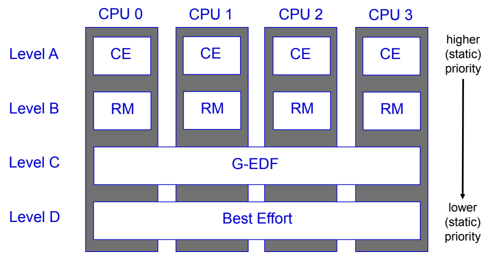


Fig. 2: Scheduling in MC^2 on a quad-core machine.

Mixed-criticality scheduling. The roots of most recent work on MC scheduling can be traced to a seminal paper by Vestal [30]. For systems where tasks of differing criticalities exist, he proposed adopting less-pessimistic execution-time assumptions when considering less-critical tasks. More formally, in a system with L criticality levels, each task has a *provisioned execution time* (*PET*)[2] specified at every level, and L system variants are analyzed: in the Level-ℓ variant, the real-time requirements of all Level-ℓ tasks are verified with Level-ℓ PETs assumed for *all* tasks (at any level). The degree of pessimism in determining PETs is level-dependent: if Level ℓ is of higher criticality than Level ℓ', then Level-ℓ PETs will generally exceed Level-ℓ' PETs. For example, in the systems considered by Vestal [30], observed WCETs were used to determine PETs for tasks at lower levels, and such times were inflated to determine PETs at higher levels.

MC^2. Vestal's work led to a significant body of follow-up work (see [6] for an excellent survey). Within this body of work, MC^2 was the first MC scheduling framework for multiprocessors (to our knowledge) [24]. MC^2 was originally designed in consultation with colleagues in the avionics industry to reflect the needs of systems of interest to them. It is implemented as a LITMUS$^{\text{RT}}$ [23] plugin and supports four criticality levels, denoted A (highest) through D (lowest), as shown in Fig. 2. Higher-criticality tasks are statically prioritized over lower-criticality ones. Level-A tasks are partitioned and scheduled on each core using a time-triggered table-driven cyclic executive.[3] Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each core.[3] On each core, the Level-A and -B tasks are required to have harmonic periods and commence execution at time 0 (this requirement can be relaxed slightly [24]). Level-C tasks are scheduled via a global earliest-deadline-first (GEDF) scheduler.[3] 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; less-deterministic computationally

[2]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.

[3]Other per-level schedulers optionally can be used, and Level-C tasks can be defined according to the sporadic task model. These options, and other considerations, such as slack reallocation, schedulability conditions, and execution-time budgeting are discussed in prior papers [12, 24, 31].

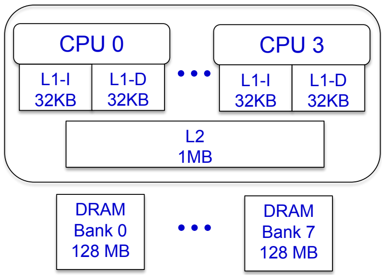


Fig. 3: Quad-core ARM Cortex A9.

intensive tasks of higher utilization would likely be assigned to Level C.

MC^2 with hardware management. In recent work [20], we developed a new MC^2 implementation that provides techniques for managing the LLC and DRAM memory banks. We briefly describe these techniques here. Our description is with respect to the machine shown in Fig. 3, 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.

In our prior work on MC^2 hardware management [20], we assumed 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). We will continue to assume this for now, but when we consider the impact of memory constraints later, we will assume that Level D is in fact present because it consumes DRAM space.

In the MC^2 variant that provides hardware management, rectangular areas of the LLC can be assigned to certain groups of tasks. This is done by using *page coloring* to allocate certain subsequences of sets (*i.e.*, rows) of the LLC to such a task group, and hardware support in the form of per-CPU *lockdown registers* to assign certain ways (*i.e.*, columns) of the LLC to the group. (Please see [20] for more detailed descriptions of these LLC allocation mechanisms.) Additionally, by controlling the memory pages assigned to each task, certain DRAM banks can be assigned for the exclusive use of a specified group of tasks. The OS can also be constrained to access only certain LLC areas and/or DRAM banks.

Fig. 4 depicts the main allocation strategy for the LLC and DRAM banks considered in our prior work [20]. This strategy ensures strong isolation guarantees for higher-criticality tasks, while allowing for fairly permissive hardware sharing for lower-criticality tasks. DRAM allocations are depicted at the bottom of the figure, and LLC allocations at the top. As seen, Level C and the OS share a subsequence of the available LLC ways and all LLC colors. (On the considered platform, each color corresponds to 128

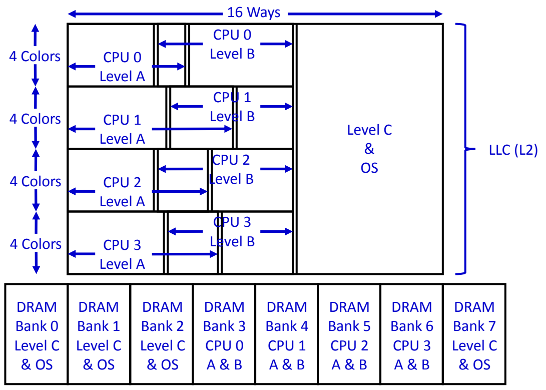


Fig. 4: LLC and DRAM bank allocation. Note that the Level-A and -B LLC areas for each core can overlap. LLC boundaries indicated by double lines are configurable parameters.

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 due to tasks on other cores (*spatial* isolation). Also, Level-A tasks (being of higher priority) do not experience LLC interference due to Level-B tasks on the same core (*temporal* isolation).

The specific number of LLC ways allocated to the Level-C/OS partition and to the per-core Level-A and -B partitions is a tunable parameter that can be determined on a per-task-set basis using optimization techniques based on linear programming presented in a prior paper [8]. These optimization techniques seek to minimize a task set's Level-C utilization while ensuring schedulability at all criticality levels.

The MC^2 implementation just described does not provide management for L1 caches, translation lookaside buffers (TLBs), memory controllers, memory buses, or cache-related registers that can be a source of contention [29]. However, we assume a measurement-based approach to determining PETs, so such unconsidered resources are implicitly considered when PETs are determined. We adopt a measurement-based approach because work on static timing analysis tools for multicore machines has not matured to the point of being directly applicable. Moreover, measurement-based methods for determining PETs are often used in practice.

Libraries and linking. In this work, we introduce support for *shared libraries* to the version of MC^2 described in the previous paragraphs. In general, programs using both shared and non-shared libraries will not contain definitions of library-provided functions and data in their source code, but will instead refer to them by name. In a post-compilation procedure known as *linking*, function and variable names are then used to locate library-provided implementations,

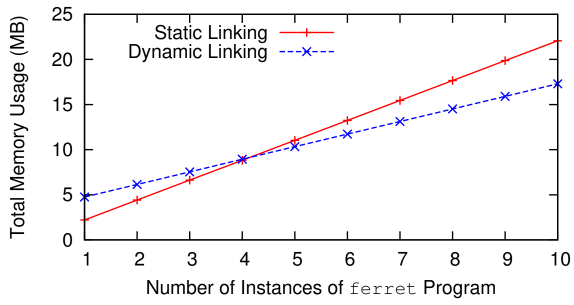


Fig. 5: Memory usage comparison of static linking and dynamic linking.

Address	KB	RSS	Mode	Mapping
00008000	960	960	r-x	ferret
000f8000	12	12	rw-	ferret
000fb000	500	500	rw-	[anon]
76f12000	528	528	rw-	[anon]
7ee0c000	620	256	rw-	[stack]
7efc1000	4	4	r-x	[anon]
ffff0000	4	0	r-x	[anon]
total	2628	2260		

(a) Static linking.

Address	KB	RSS	Mode	Mapping
00008000	136	136	r-x	ferret⋆
00031000	4	4	r--	ferret
00032000	4	4	rw-	ferret
00033000	488	488	rw-	[anon]
76b49000	520	520	rw-	[anon]
76bcb000	872	872	r-x	libc-2.19.so⋆
76cac000	8	8	r--	libc-2.19.so
76cae000	4	4	rw-	libc-2.19.so
76caf000	12	12	rw-	[anon]
76cb2000	100	100	r-x	libgcc_s.so.1⋆
76cd2000	4	4	rw-	libgcc_s.so.1
76cd3000	396	396	r-x	libm-2.19.so⋆
76d3d000	4	4	r--	libm-2.19.so
76d3e000	4	4	rw-	libm-2.19.so
76d3f000	296	296	r-x	libjpeg.so.7.0.0⋆
76d90000	4	4	r--	libjpeg.so.7.0.0
76d91000	4	4	rw-	libjpeg.so.7.0.0
76d9e000	152	152	r-x	libgslcblas.so.0.0.0⋆
76dcb000	4	4	r--	libgslcblas.so.0.0.0
76dcc000	4	4	rw-	libgslcblas.so.0.0.0
76dcd000	1396	1396	r-x	libgsl.so.0.17.0⋆
76f31000	8	8	r--	libgsl.so.0.17.0
76f33000	56	56	rw-	libgsl.so.0.17.0
76f41000	92	92	r-x	ld-2.19.so⋆
76f59000	28	28	rw-	[anon]
76f60000	4	4	r--	ld-2.19.so
76f61000	4	4	rw-	ld-2.19.so
7ecce000	620	256	rw-	[stack]
7eecf000	4	4	r-x	[anon]
ffff0000	4	0	r-x	[anon]
total	5236	4868		

⋆ This mapping can be shared by several tasks.

(b) Dynamic linking.

TABLE I: Task memory map with **(a)** static vs. **(b)** dynamic linking.

which are then used to produce the fully functioning final program. Using libraries in this way can eliminate the need to recompile commonly used functions, as library code can be compiled once and left unchanged as a program is developed. The standard library for the C programming language is a classic example of a library in widespread use, and contains commonly used functions including `printf` and `strnlen`.

In Linux and other modern operating systems, libraries containing compiled code may be either *static libraries* or *shared libraries*. Static and shared libraries differ in both how they are allocated in memory and the time at which linking occurs.[4]

Static libraries. Compiled code and data from static libraries are merged into a complete executable program file in a process called *static linking*. On systems such as Linux, this means that page frames in physical memory are needed for both the base program and any content copied from static libraries. If the same library (*e.g.*, the C standard library) is used by many or all programs, *many copies of the library code are created, increasing physical memory consumption*.

Shared libraries. References to shared libraries are resolved using *dynamic linking*. Dynamic linking occurs either at load time, before a task (*i.e.*, process) begins execution, or at runtime when a library is first referenced. Dynamic linking requires the shared library to be mapped into the task's virtual address space. Unlike under static linking, where library copies necessarily reside in local memory, the read-only portions of shared libraries such as executable code can be shared between many processes. So, while shared libraries may increase a single process' memory requirements, they still reduce memory requirements of the overall system.

Memory usage comparison. Even though tasks using static libraries may individually have smaller memory footprints, in aggregate the use of dynamic libraries typically saves memory. To illustrate the potential memory savings provided by shared libraries, we will refer to Fig. 5 and Tbl. I. Fig. 5 shows total memory usage of multiple instances of the `ferret` program from the PARSEC benchmark suite [5], and Tbl. I shows memory regions reported by the `pmap` tool for the `ferret` program. The first column shows the virtual address of each allocated memory region, the second column shows the size of the allocated virtual memory in KB, and the third column shows the size of the allocated physical memory, called the *resident set size* (*RSS*), in KB. The fourth column shows access permissions: read, write, and execute. The last column indicates a file name for file-backed mapping, `[ anon ]` for non-file-backed ("anonymous") allocations, or `[ stack ]` for the program stack. Tbl. I(a) shows the simpler memory map for a statically linked version of `ferret`, and Tbl. I(b) shows the memory map when shared libraries are used instead. As seen in Fig. 5, if five or more instances of the `ferret` program are running in a system, then dynamic linking saves memory by sharing the libraries listed in Tbl. I(b).

In our previous work [7, 20], memory was assumed to be an unconstrained resource when checking schedulability. In real systems, however, memory *is* a constrained resource,

[4]During the initialization procedure for real-time tasks in LITMUS$^{\mathrm{RT}}$, the `mlockall()` function is invoked to allocate and map page frames for all program and library virtual addresses, populate their content, and pin them in memory.

especially in application domains with the need to support multiple modes of operation. For example, current avionics software can have over a dozen operating modes, and envisioned unmanned aircraft could require many more. As an example, consider a system with 20 modes and 25 distinct tasks per mode.[5] Assuming the allocations shown in Tbl. I for each task, the total memory usage would be 1103.52 MB (2260 KB $\cdot$ 25 $\cdot$ 20) with static linking and 700.63 MB (1428 KB $\cdot$ 25 $\cdot$ 20 + 3440 KB) with dynamic libraries. Thus, assuming the target hardware platform is that shown in Fig. 3, static linking would not be tenable, because that platform provides only 1GB of DRAM.[6]

III. Implementation

Statically linked programs contain *non-shared* copies of library code in memory. To save memory, we propose an alternative strategy, which we refer to as *selective sharing*, that enables us to take advantage of dynamic linking's memory savings. Informally, selective sharing requires regulating "who" may share "what" with "whom." In devising the selective-sharing approach, our major objective was to reduce the system's memory footprint while *preserving all pre-existing isolation properties of MC^2*. The schedulability improvements afforded by selective sharing when DRAM constraints are considered are discussed in Sec. IV.

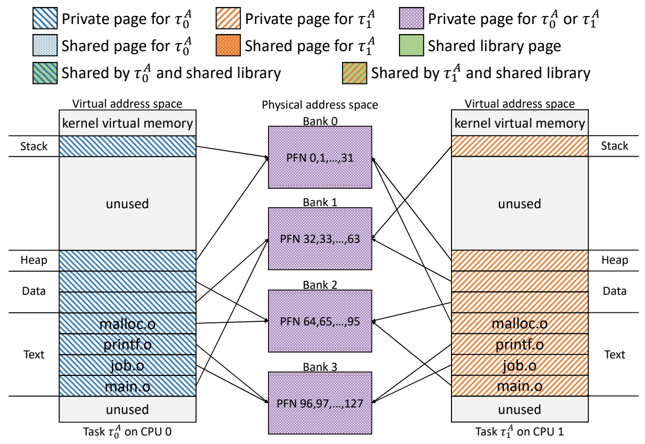
We begin this section by describing how unrestricted shared library usage breaks hardware isolation. We follow this with a description of selective sharing and compare several implementation strategies. We note that any implementation of selective sharing will yield the same schedulability benefits, so the discussed approaches only differ in their relative ease-of-implementation.

Hardware isolation with shared libraries. To see why shared libraries are problematic when providing hardware isolation, consider Fig. 6, which depicts the virtual address spaces for two Level-A tasks, τ_0^A and τ_1^A, on a dual-core machine with four DRAM banks, each of which contain 32 pages of physical memory. This is a simpler machine than our Cortex-A9 test platform shown in Figs. 3 and 4, but it is sufficient to illustrate the effects of hardware interference. In Fig. 6, physical memory pages are referred to by page frame numbers (PFNs), and the PFNs in each row of a DRAM bank correspond to a single color in the LLC.

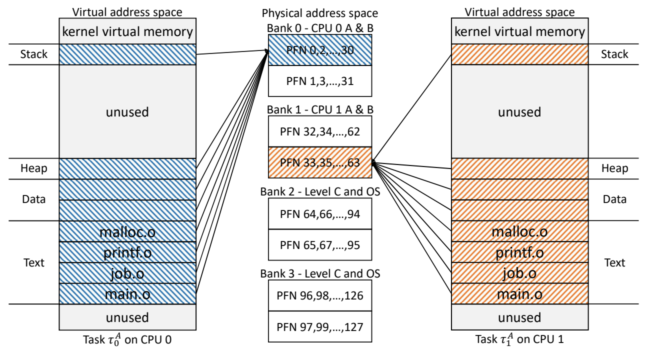
Fig. 6 (a) shows an example virtual-to-physical address mapping of statically linked programs before our isolation techniques are applied. The figure shows task τ_0^A executing on CPU 0 and task τ_1^A executing on CPU 1. In Fig. 6 (a), these tasks can use arbitrary page frames, as represented by the lines connecting virtual-memory regions to arbitrary DRAM regions. This can result in LLC or bank interference

[5]This is a simple example used to illustrate a point; it does not consider the OS and the fact that different modes may have tasks in common.

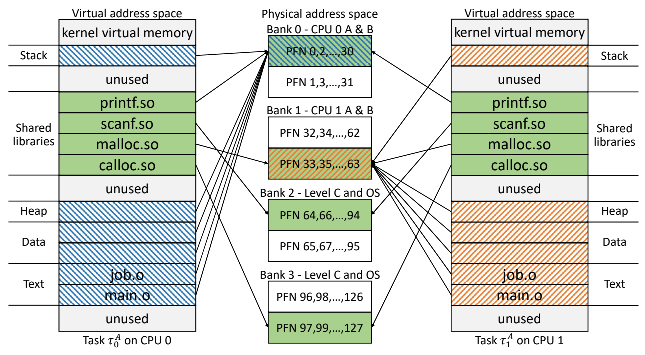
[6]Though not strictly required, it is desirable in avionics systems for all tasks of all modes to be memory-resident if possible, to avoid unexpected mode-transition delays.



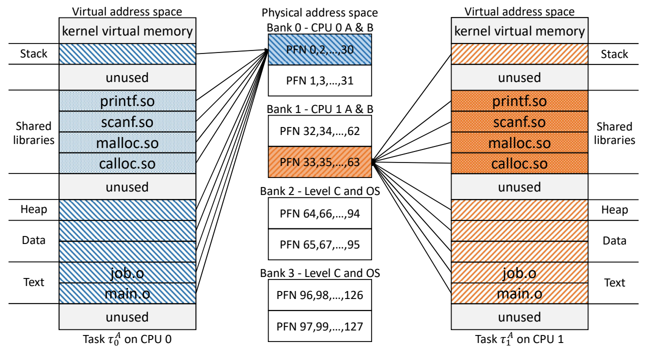
(a) Unmanaged LLC and DRAM with static linking.



(b) Hardware isolation with static linking.



(c) Hardware isolation with shared libraries.



(d) Hardware isolation with selective sharing.

Fig. 6: Virtual address space and mappings to page frames and LLC under MC^2.

if the two tasks attempt to access the same DRAM bank or pages mapped to the same LLC region.

In our prior work on MC^2 with hardware management, we provided hardware isolation by remapping page frames [20]. This remapping was realized by introducing a page-coloring system call to MC^2 that each real-time task invokes in order to properly color its pages (in accordance with the scheme depicted in Fig. 4). After the system call completes, the task only uses specific LLC areas and DRAM banks. Fig. 6 (b) illustrates this: task τ_0^A uses entirely separate DRAM banks and page colors from task τ_1^A. Now, the two tasks can run concurrently without having to guard against interferring with one another in the LLC or DRAM banks.

However, as we discussed in Sec. II, this approach requires static linking and consumes more DRAM space, leading to degraded schedulability when memory is considered as a constrained resource. If we wish to save memory by allowing shared libraries, then the mechanisms used previously for providing hardware isolation with static linking are no longer sufficient. In particular, as illustrated in Fig. 6 (c), non-shared page frames can still be isolated, but pages belonging to shared libraries are accessible to any task that uses the library, even tasks on different CPU cores. (*e.g.*, invocations of `printf()` by the two tasks under consideration map to the same DRAM bank). For this reason, we disallowed shared pages in our prior work.

Selective sharing. Selective sharing preserves the notion of isolation provided by MC^2 while more efficiently utilizing memory by introducing *per-partition library replicas*. Specifically, up to m+2 replicas are created per library, where m is the number of cores: one per-core copy is shared by all Level-A and -B tasks running on that core, another copy is shared by Level-C tasks on all cores, and an additional copy is used by non-real-time background services. We replicate only the shared read-only sections of a library, which hold both instructions and read-only data. The writable data sections of shared libraries are private, and therefore handled along with other private pages by the page-coloring system call from our prior work.

Fig. 6 (d) illustrates the impact of selective sharing. In this particular example, a replica of each shared library exists for each CPU and is allocated in the CPU's DRAM partition. As seen in Fig. 6 (d), a library loaded by task τ_0^A will be allocated in Bank 0, and a library loaded by task τ_1^A will be allocated in Bank 1. Furthemore, the allocations will use pages of different colors in order to map to different LLC regions. This ensures that MC^2's isolation properties are maintained.

Having described the concept of selective sharing, it remains to discuss whether it can be efficiently implemented. There are actually several possible implementation strategies, three of which we discuss next. These strategies vary in implementation complexity, but all require some kernel support for providing isolation in DRAM banks and the LLC.

Creating on-disk replicas of shared libraries. The simplest way to create per-partition library replicas is to use Linux's `LD_LIBRARY_PATH` environment variable, which adds a new list of directories the dynamic linker searches for shared libraries before it searches the default directory list. To use this approach, a user must create copies of shared libraries on disk and place them in separate directories. Then, before each real-time task is created, the user must set `LD_LIBRARY_PATH` for each specific task so that it refers to the set of shared libraries corresponding to its criticality level and core assignment (if it is a Level-A or -B task). This approach has the advantage of requiring less kernel support (kernel support is still required to implement page coloring), but it requires nontrivial maintenance, disk usage, and configuration effort by the user. These disadvantages become more pronounced if a large number of shared libraries are in use.

Modifying the dynamic linker to provide selective sharing. A modified dynamic linker can potentially provide selective sharing while alleviating some of the maintenance difficulties associated with on-disk replicas. The dynamic linker is responsible for mapping shared libraries into each process's address space, and typically does so by directly memory-mapping shared library files. A dynamic linker capable of selective sharing would instead need to allocate separate regions of shared memory for each library replica, based on the current task's criticality level and core assignment (if it is a Level-A or -B task).

While this approach would be easier to use than on-disk replication, it poses greater implementation challenges. This approach would not only require modifying the code comprising the GNU dynamic linker,[7] but would still require more kernel support than on-disk replicas using `LD_LIBRARY_PATH`. Specifically, it would require a memory-allocation system capable of providing a page from a specific allocation pool to the dynamic linker, and the new allocation system must not interfere with existing MC^2 code for coloring non-shared pages. Due to this reason, and our group's relative unfamiliarity with the GNU dynamic linker code base, we did not attempt to implement this approach. Even so, we believe that an implementation in this manner is possible.

A transparent kernel-level selective sharing implementation. The final approach we discuss is transparent to users and is of similar difficulty to, or easier than, dynamic linker modification. This kernel-only approach requires extending the page-coloring system call from our previous work on MC^2 in order to specially handle shared-library pages. The final result is a system call which identifies, replicates, and migrates shared-library pages to the correct partitions in physical memory, in addition to maintaining previous MC^2 functionality.

[7]The dynamic linker used by most Linux distributions is provided by GNU libc, and consists of several thousand lines of code.

The Linux kernel offers built-in page migration functions that are capable of safely relocating physical pages.[8] We were able to use these functions in previous work to assist with coloring non-shared memory, but unfortunately the default behavior of these functions makes them unusable for remapping shared pages. We worked around this limitation by tracking shared-library pages in a separate data structure and manually overriding the kernel's page-migration behavior when necessary. Beyond userspace transparency and disk-space savings, this approach has the advantage of not changing existing system behavior: only programs that invoke the system call are affected. We note again that, without changing Linux's existing memory-allocation system, kernel modification for migrating shared pages to provide MC^2's isolation properties is still required for any approach to library replication. The complete code for providing kernel-level selective sharing, page coloring, and the associated userspace library is available online.[9]

Memory footprint statistics. To illustrate the impact of selective sharing on DRAM use, we created a simple task system, using publicly available benchmark programs in the Data Intensive Systems (DIS) stressmark suite [25] and the PARSEC benchmark suite [5] and analyzed DRAM consumption. Tbl. II presents the RSS of the considered benchmark programs when libraries were non-shared (*i.e.*, linked statically) and selectively shared (*i.e.*, linked dynamically). This data was computed assuming one instance of each of tasks $\tau_1, \ldots, \tau_{10}$ is assigned to Levels A and B on each core and one instance of each of tasks τ_{11} and τ_{12} is assigned to Level C, on the hardware platform shown in Figs. 3 and 4. This task system requires 11.7% less memory if selective sharing is used. The relative impact of selective sharing increases if a greater opportunity for sharing exists. For example, if we were to increase to five instances of each of $\tau_1, \ldots, \tau_{10}$ per core and five instances of each of τ_{11} and τ_{12}, then memory consumption would be reduced by 22.9% compared to static linking.

While these results demonstrate that our techniques can improve efficiency in DRAM use, understanding the benefits in a holistic sense requires examining impacts on overall schedulability. To assess such impacts, we conducted a large-scale overhead-aware schedulability study. Before discussing the results of this study, we first delve into some issues that arose when introducing the effects of memory-capacity limits on schedulability.

IV. Impact of Introducing Memory Constraints

Understanding the impacts of different methods for supporting shared libraries in MC^2 requires an understanding of the limitations on available DRAM space on our studied platform. In this section, we provide more detail regarding this issue. We also examine DRAM and LLC allocation

Task	Program Name	Non-Shared	Selectively Shared	
			Private Pages	Shared Pages
τ_1	field	1456 KB	1136 KB	4920 KB
τ_2	update	2284 KB	1964 KB	
τ_3	matrix	1936 KB	1600 KB	
τ_4	transitive	680 KB	360 KB	
τ_5	neighborhood	1208 KB	852 KB	
τ_6	pointer	2284 KB	1964 KB	
τ_7	blackscholes	700 KB	340 KB	
τ_8	ferret	2260 KB	1428 KB	
τ_9	swaptions	1064 KB	392 KB	
τ_{10}	x264	1368 KB	348 KB	
τ_{11}	fluidanimate	10328 KB	9720 KB	2144 KB
τ_{12}	freqmine	23824 KB	23188 KB	

TABLE II: DRAM consumption of an example task system assuming non-shared and selectively shared libraries.

techniques disregarded in our prior work. These techniques break certain isolation guarantees and thus were *pointless to consider earlier*. However, as we will show, these techniques provide advantages in DRAM space allocation and so *are worth considering now* since we are viewing the available DRAM space as a constrained resource in this work.

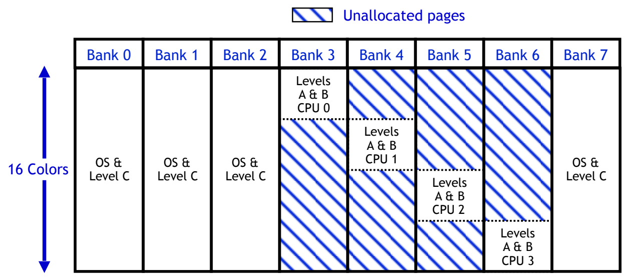
Cortex A9 DRAM allocation. Fig. 7 (a) shows a more detailed view of the DRAM allocation scheme assumed in our prior work [20], as depicted earlier in Fig. 4. Levels A and B are bank-partitioned from other cores to avoid cross-core contention in bank access at the highest criticality levels. Additionally, *bank interleaving*, which spreads contiguous pages across multiple banks, is disabled. This is because, under bank interleaving, each bank only contains two page colors, which creates dependencies between LLC allocation and bank allocation. When bank interleaving is disabled, each bank has pages of all 16 colors, which allows banks and colors to be allocated independently.

Unfortunately, with DRAM now being viewed as a constrained resource, a disadvantage of disabling bank interleaving is exposed: because each Level-A/B area of the LLC corresponds to one quarter of the available colors (refer to Fig. 4), only one quarter of the pages in each Level-A/B bank can be allocated (refer to Fig. 7 (a)). One way to reclaim this lost DRAM space while maintaining LLC isolation is to enable bank interleaving and assign to the Level-A/B subsystem on each core an LLC area corresponding to the colors of the designated bank for that core. Fig. 7 (b) depicts the resulting DRAM allocations and Fig. 8 shows the corresponding LLC layout. In this paper, we consider this *interleaved approach* as an alternative to the *non-interleaved approach* considered in our prior work.

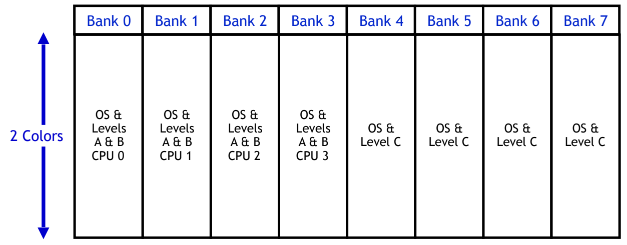
The new interleaved approach being considered here comes at a cost. As shown in Fig. 7 (b), the OS is no longer restricted to DRAM banks shared with Level C. Specifically, the OS claims the first several hundred pages of physical memory at boot time.[10] These pages are used for unmovable memory, including the kernel image and the area reserved for `ZONE_DMA` memory used by certain

[8]https://www.kernel.org/doc/Documentation/vm/page_migration
[9]https://wiki.litmus-rt.org/litmus/Publications

[10]This is one of the reasons why the Level-C/OS DRAM banks in Fig. 7 (a) are not contiguous.



(a) Non-interleaved.



(b) Interleaved.

Fig. 7: Page allocation in DRAM banks.

hardware devices. While these pages are relegated to Bank 0 when interleaving is disabled, they are spread across all banks when interleaving is enabled. As a result, interleaving eliminates bank isolation from the OS at Levels A and B. Additionally, because Level C is now color partitioned in the LLC (refer to Fig. 8), the OS (which has access to all page colors) can no longer be restricted to the Level-C LLC partition. In order to ensure that Levels A and B are isolated from the OS in the LLC, we assume that under the interleaved scheme, the OS simply bypasses the LLC.

Another way to maintain OS isolation in the LLC is to provide the OS with a subset of the LLC ways. Additionally, other viable approaches may exist for reclaiming unused DRAM space. However, to keep the study presented herein at a manageable level, it is not possible to analyze every possible variant of our basic allocation schemes. We defer a full consideration of all possible variants to future work.

From the above discussion, it should be clear that both allocation schemes under consideration have advantages and disadvantages. To better understand the tradeoffs between them, we included both in our overhead-aware schedulability study, which is discussed in Sec. V.

Support for modes and Level D. To this point, we have ignored Level D. Level D can be incorporated into both considered allocation schemes by requiring Level-D tasks to execute within the same DRAM banks and LLC area(s) as Level-C tasks. This approach has no major implications with respect to any isolation properties afforded.[11] With respect to schedulability, however, there is one impact: the available DRAM capacity for Level C is reduced. Further reductions

[11]Introducing Level D would likely have little effect on Level-C tasks' Level-C PETs, which are based on the average case (recall Sec. II).

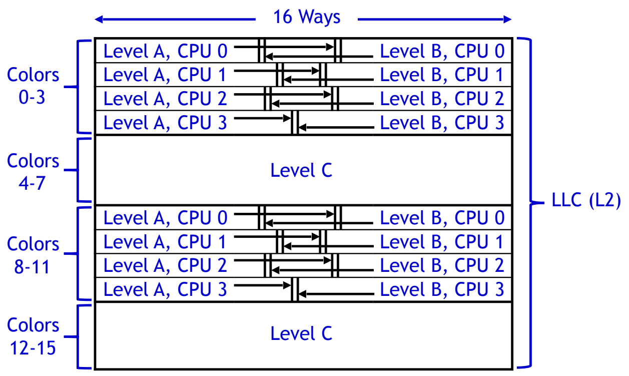


Fig. 8: Interleaved LLC allocation.

in capacity might occur at all levels, due to the need to support multiple modes, as discussed at the end of Sec. II. Because of these sources of DRAM capacity reduction, in the experiments presented in Sec. V, we consider scenarios in which the full capacity of DRAM might not be available.

With respect to mode changes, Fig. 7 (a) suggests an interesting possibility to explore in future work: using the unallocatable DRAM space in Banks 3–6 to support tasks of other modes. However, each mode would then have a different color allocation for its Level-A and -B tasks on each core. Producing such an allocation could be difficult, because in reality, different modes typically have some tasks in common. Instead of delving into such complex issues in this paper, we simply assume that under the non-interleaved scheme, all modes must share the same per-core allocation of colors—or equivalently, when analyzing a given task system for schedulability, some fraction of its allocated DRAM space may simply be "lost" due to tasks of other modes.[12]

V. Evaluation

To assess the efficacy of selective sharing, we evaluated the schedulability of millions of randomly generated task systems under the MC^2 variants listed in Tbl. III. In denoting these variants, the prefix "I" (resp., "NI") denotes that the interleaved (resp., non-interleaved) memory-allocation scheme was used, as depicted in Fig. 7 (b) and Fig. 8 (resp., Fig. 7 (a) and Fig. 4). The suffix used indicates how libraries were dealt with: "STC" denotes statically linked libraries, "SSH" denotes selectively shared libraries (as proposed herein), and "IDL" (ideal) denotes viewing memory as an unconstrained resource (as we did in our prior work). In addition to these MC^2 variants, we considered the HRT uniprocessor earliest-deadline-first scheduler, denoted U-EDF. This reflects current industry practice for eliminating shared-hardware interference by disabling all but one core. Under U-EDF, no DRAM capacity constraints were assumed.

[12]Issues related to mode-change semantics are beyond the scope of this paper. Mode changes are being considered here only with respect to lost DRAM capacity due to having to support tasks required in other modes.

DRAM & Linking Assumptions	DRAM-Aware		DRAM-Oblivious
	Static	Sel. Shared	DRAM Constraints Ignored
Interleaved	I-STC	I-SSH	I-IDL
Non-Interleaved	NI-STC	NI-SSH	NI-IDL

TABLE III: Considered MC^2 variants.

Under the “STC” and “SSH” schemes, constraints on DRAM memory were considered. For these schemes, we calculated the total number of DRAM pages available to real-time tasks in MC^2 on our considered ARM platform after accounting for pages used by the OS. For the I-STC and I-SSH schemes, Levels A and B have approximately 25,000 pages allocated per core, and Level C has approximately 100,000 pages allocated in total. For the NI-STC and NI-SSH schemes, Levels A and B have approximately 8,000 pages allocated per core, and Level C has approximately 73,000 pages allocated in total. Under the I-SSH and NI-SSH schemes, a task system’s DRAM consumption was calculated assuming that all libraries are selectively shared, as described in Sec. III.[13]

Task-system generation. We generated task systems at random by extending the process used in our prior work [20] to account for DRAM consumption. As explained in detail in [20], PETs were determined at Level C (resp., Level B) based on measured average-case (resp., worst-case) execution-time data; Level-A PETs were obtained by applying a 50% inflation factor to Level-B PETs.

Task systems were randomly generated by using seven uniform distributions to choose task and task-system parameters. The specific distributions used were selected from the per-distribution choices listed in Tbl. IV. These distributions are defined with respect to the U-EDF scheme. All combinations of these choices were considered. These distributions determine the criticality utilization ratio (*i.e.* the fraction of the overall utilization assigned to each criticality level), task periods, task utilizations, and the maximum LLC reload time after a preemption or migration (specified as a fraction of overall task execution time). At a high level, our overall experimental framework refines the following step-wise process used in our prior work [20]:

Step 1: Select seven specific distributions from among the distribution categories listed in Tbl. IV.

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

Step 3: Based on the generated U-EDF PETs, generate PETs for the MC^2 schemes. This process is informed by micro-benchmark data, as discussed at length in [20].

Step 4: Adjust the generated task parameters to account for relevant overheads. As discussed in detail in [20], the actual overhead values applied are based on measured data.

[13]To keep our study manageable, we examined the two extremes of always using static linking and always using selective sharing. In practice, some combination of the two might result in the best memory utilization.

Category	Choice	Level A	Level B	Level C
1: Criticality Utilization Ratios	A-Heavy	[50, 70)	[10, 30)	[10, 30)
	B-Heavy	[10, 30)	[50, 70)	[10, 30)
	C-Heavy	[10, 30)	[10, 30)	[50, 70)
	AB-Mod.	[35, 45)	[35, 45)	[10, 30)
	AC-Mod.	[35, 45)	[10, 30)	[35, 45)
	BC-Mod.	[10, 30)	[35, 45)	[35, 45)
	All-Mod.	[35, 45)	[35, 45)	[35, 45)
2: Period (ms)	Short	{3, 6}	{6, 12}	[3, 33)
	Long	{48, 96}	{96, 192}	[50, 500)
3: Task Utilization	Light	[0.001, 0.03)	[0.001, 0.05)	[0.001, 0.1)
	Heavy	[0.1, 0.3)	[0.2, 0.4)	[0.4, 0.6)
4: Max Reload Time	Light	[0.01, 0.1)	[0.01, 0.1)	[0.01, 0.1)
	Heavy	[0.25, 0.5)	[0.25, 0.5)	[0.25, 0.5)
5: % DRAM Reserved for Task Set	Small	10		
	Moderate	40		
	Large	80		
6: Priv. Page Count (SSH) in Hundreds	Light	[0.5, 2)	[0.5, 2)	[1, 4): 0.75 [5, 10): 0.25
	Heavy	[2, 5)	[2, 5)	[3, 6): 0.75 [10, 70): 0.25
7: Priv. Page Count Inc.	Light	[80, 150)	[80, 150)	[100, 300)
	Heavy	[150, 250)	[150, 250)	[300, 500)

TABLE IV: Task-set parameters and distributions. In Category 6, last column, I:P denotes that interval I is selected with probability P.

Name	Size in Kilobytes
libc	872
ld	92
librt	20
libpthread	64
libm	396

Name	Size in Kilobytes
libstdc++	608
libjpeg	296
libgslcblas	152
libgsl	1396
libgcc_s	100

TABLE V: The size of the code segments of considered libraries under selective sharing.

Step 5: For each task, assign linked libraries from the list in Tbl. V. These are libraries used by the benchmark programs listed in Tbl. II. The first four libraries, which are shared by all benchmark programs under MC^2, were assigned to all tasks. *Libm* was randomly assigned to Level-A and B tasks. The remaining libraries are used by computationally intensive benchmark programs and were only assigned to Level-C tasks. To each Level-C task, we randomly assigned two to six libraries from the last six libraries listed.

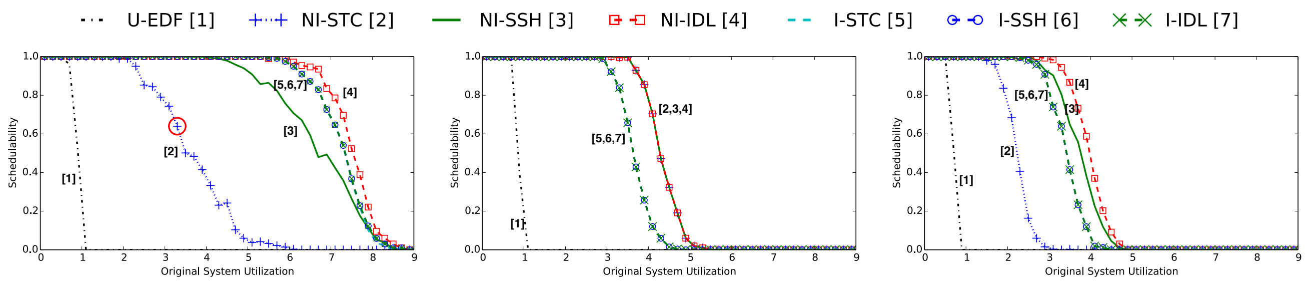
Step 6: Assign Level-A and -B tasks to cores using a worst-fit decreasing heuristic as discussed in [20].

Step 7: Determine the DRAM consumption of the task system under the “STC” and “SSH” schemes. This step is described in detail in an appendix. (The distributions in Category 6 and 7 are used here.)

Step 8: Test the schedulability of the resulting task system under U-EDF and each MC^2 scheme in Tbl. III. This step is also described in detail in an appendix.

The distributions in Tbl. IV were defined to enable the systematic study of different factors impacting schedulability, such as MC analysis, shared-libary usage, and DRAM constraints. Moreover, the distribution choices in Tbl. IV 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



(a) (C-Heavy, L., H., L., S., L., H.) (b) (C-Heavy., L., H., L., L., L., L.) (c) (B-Heavy, L., H., H., S., L., H.)

Fig. 9: Representative schedulability plots.

using a tuple notation. For example, (C-Heavy, Long, Heavy, Light, Moderate, Heavy, Light) denotes using the C-Heavy, Long, Heavy, *etc.*, distribution choices in Tbl. IV. 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 2,000 task systems.

Schedulability results. In total, we evaluated the schedulability of over five million randomly generated task systems, which took roughly 25 CPU-days of computation. From this abundance of data, we generated 672 schedulability plots, of which three representative plots are shown in Fig. 9. The full set of plots is available online [19].

Each schedulability plot corresponds to a single scenario. To understand how to interpret these plots, consider Fig. 9(a). In this plot, the circled point indicates that 64% of the generated task systems with U-EDF utilizations of 3.3 were schedulable under the NI-STC scheme. Note that, because the x-axis represents system utilizations under the single-core HRT U-EDF scheme, it is possible under MC^2 to support systems with a 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. 9.

Obs. 1. Schedulability under I-STC was better than under NI-STC in 61% of cases. Conversely, schedulability under NI-SSH was better than under I-SSH in 54% of cases.

This observation is supported by insets (a) and (c) of Fig. 9. Under the "STC" schemes, DRAM is more of a limiting resource (since static linking wastes space), so the DRAM loss illustrated Fig. 7 (a) for non-interleaved memory becomes a liability. Under the "SSH" schemes, DRAM is a less-constraining resource, so the virtues of using non-interleaved memory usually win out (it is these virtues that caused us to only consider this possibility in our prior work).

Obs. 2. Schedulability loss under I-STC (resp., NI-STC) was non-negligible (at least 10% of one core's capacity compared to an "IDL" allocation) in 27% (resp., 61%) of scenarios.

Fig. 9 (b) shows a scenario with negligible loss. In scenarios with light memory consumption or large DRAM reservations, schedulability was rarely impacted by DRAM capacity.

Obs. 3. For scenarios with non-negligible schedulability loss under I-STC (resp., NI-STC), I-SSH (resp., NI-SSH) regained on average 43% (resp., 36%) of schedulability lost under I-STC (resp., NI-STC).

Such regained schedulability can be seen in insets (a) and (c) of Fig. 9. (Note that it is unreasonable to expect most of the loss to be consistently regained, because the "IDL" schemes may deem systems to be schedulable whose memory footprints will not even fit into DRAM regardless of libraries.) From these and similar scenarios, we conclude that selective sharing, as proposed in this paper, can result in significant schedulability gains.

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

VI. Prior Related Work

This work follows a long line of research examining shared-resource contention in real-time systems [21]. Prior efforts have focused on issues such as cache partitioning [3, 13, 17, 32], DRAM controllers [4, 14, 15, 22], and bus-access control [1, 2, 9, 10, 11, 27]. Other work has focused on reducing shared-resource interference when per-core scratchpad memories are used [28], accurately predicting DRAM access delays [16], throttling lower-criticality tasks' memory accesses [34], allocating memory [33], and enhancing the temporal isolation by managing shared pages [18]. In one of these papers [18], the management of shared pages to prevent timing penalties caused by the eviction of shared pages is considered, but that work does not consider interference due to contention for shared-hardware resources.

To our knowledge, we are the first to consider in detail the unique impact that sharing memory has on hardware isolation under the notion of MC scheduling espoused by Vestal [30], which was proposed with the express intent of *achieving better platform utilization*. Several of the aforementioned papers do target MC systems [4, 9, 10, 11, 14, 15, 22, 26, 34], but only peripherally touch on the issue of achieving better platform utilization, if at all. Also, most of them focus on hardware design. One of these papers [1] considers systems in which tasks share data, but does not consider the specific impact this has on hardware isolation. Hardware isolation under Vestal's notion of MC scheduling is the subject of five prior MC^2-related papers by our group [7, 12, 20, 24, 31]. One of these papers [20] was reviewed in detail in Sec. II; we refer the reader to [20] for an overview of the remaining three [12, 24, 31].

Issues related to problems caused by sharing in the context of MC^2 were also considered in a prior paper by our group [7], but that work dealt with sharing that arises from the usage of shared data buffers. That source of sharing introduces very different concerns than do shared libraries. First, shared data buffers introduce read/write sharing, while shared libraries introduce read-only sharing. Second, it is not possible to completely eliminate shared pages due to the usage of shared data buffers. Thus, in some sharing scenarios (*e.g.*, sharing between a Level-A task and a Level-C task), the isolation properties afforded by MC^2 are fundamentally compromised. As a result, [7] mainly focuses on techniques that ameliorate (rather than entirely eliminate) the detrimental impacts of data sharing. In contrast, as we have seen, shared libraries can be supported under MC^2 without violating any of MC^2's isolation properties. Finally, the major issue with shared libraries is schedulability loss due to memory-capacity constraints when libraries are replicated. Such constraints were not considered at all in [7]. Indeed, shared pages due to the usage of shared data buffers cannot be elminated through replication or any other means.

VII. Conclusion

Most proposed approaches for controlling cross-core interference on multicore platforms rely on techniques that isolate tasks with respect to memory-related shared-hardware accesses. In practice, however, tasks commonly share memory pages. Thus, for any management approach to have practical impact, the issue of sharing, which directly breaks isolation, must be addressed. In a recent paper [7], we examined this issue as it pertains to shared data buffers. In this paper, we have examined it with respect to shared libraries.

Shared libraries are an issue because memory is a constrained resource. Thus, devising an approach for handling libraries prompted us to consider the schedulability-related impacts of memory limits in our work on MC^2 for the first time. Such a consideration led us to examine alternative approaches for allocating DRAM and LLC space that were pointless to consider before. We evaluated the considered allocation approaches via a large-scale overhead-aware schedulability study. In this study, our approach for dealing with shared libraries often improved schedulability significantly.

The results of this paper suggest many avenues for further research. First, to simplify the presentation in this paper, we chose to consider an earlier variant of MC^2 that preceded that which supports shared data buffers [7]. In future work, we intend to extend the study presented herein by considering such buffers in addition to libraries. Second, alternative strategies for allocating DRAM and LLC space were mentioned in Sec. IV. We intend to evaluate all of these strategies in future work. Finally, we have noted several times that the need to support multiple modes can further constrain memory. Adding support for mode-change protocols to MC^2 is actually a non-trivial issue that warrants further study.

Acknowledgment

We thank Prakash Sarathy of Northrop Grumman for answering questions about avionics use cases. We are also grateful to the anonymous referees for providing many helpful suggestions.

References

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

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

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

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

[5] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

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

[7] 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*.

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

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

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

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

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

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

[14] 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*.

[15] 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*.

[16] 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*.

[17] 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*.

[18] H. Kim and R. Rajkumar. Memory reservation and shared page management for real-time systems. *Journal of Systems Architecture*,

60:165–178, Feb. 2014.

[19] N. Kim, M. Chisholm, N. Otterness, J. Anderson, and F.D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. Full version of this paper, available at http://jamesanderson.web.unc.edu/papers/.

[20] 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*.

[21] 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*.

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

[23] LITMUS^{RT} Project. http://www.litmus-rt.org/.

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

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

[26] 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*.

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

[28] 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*.

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

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

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

[32] 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*.

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

[34] 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*.

Appendix: Additional Details Concerning the Task-System Generation Process

In this appendix, we provide further details concerning Steps 7 and 8 of the task-system generation process described in Sec. V.

Step 7: To determine the DRAM consumed by a task system at Level C and with respect to each Level-A/B bank, we first must calculate the DRAM consumption for private (*i.e.*, non-shared) pages of tasks. For each task, we choose a private-page count from the selected distribution under Category 6. This is the private-page count under the "SSH" schemes. The distributions chosen reflect private-page counts for the considered benchmark tasks in Tbl. II when all libraries are selectively shared. In keeping with our thesis that Levels A and B consist mostly of "fly-weight" tasks (refer to Sec. II), page-count distributions for Level-A and -B tasks yield smaller page counts than those for Level-C tasks.

We add an additional number of private pages under the "STC" schemes. The page-count increase for a task is chosen from the selected distribution in Category 7. Page-count increases under static linking for the benchmark programs in Tbl. II are within the range [80, 260). Category-7 distributions were designed to encompass this range. Real-world systems may use more library functions than used in our benchmark programs, which could lead to greater static-linking overheads. To account for this possibility, we actually allow page-count increases up to 500. To ensure that the page-count increase for a task is reasonable, we adjust the increase to be no greater than 90% of the combined size of all libraries linked to the task.

Note that the maximum LLC reload time after a preemption or migration is dependent on the number of addresses a task uses. We adjust the max LLC reload time of each task to be no greater than the time required to load into the LLC all addresses the task consumes in DRAM.

The total DRAM consumed by a task system in a given core's Level-A/B DRAM bank is the DRAM consumed by all Level-A/B tasks assigned to that core plus, under the "SSH" schemes, the DRAM consumed by any libraries selectively shared by such tasks on that core. The DRAM consumed by Level C across all Level-C banks is the DRAM consumed by all Level-C tasks plus, under the "SSH" schemes, the DRAM consumed by libraries selectively shared at Level C.

Step 8: For the MC^2 schemes, we adapted the optimization scheme presented by us previously [8, 20], which involves solving a mixed-integer linear programs (MILP), to determine LLC allocations while ensuring that all MC^2 schedulability conditions (defined in [24]) are met.[14] We added constraints to these schedulability conditions that ensure that DRAM capacity constraints are met. These constraints are applicable to the schemes denoted "DRAM-Aware" in Tbl. IV.

For these schemes, if the DRAM consumed in a Level-A/B bank or at Level C exceeds the DRAM *reserved* for the task system in any bank or at Level C, then we deem the system as unschedulable under that scheme. *Reserved DRAM* is the portion of available DRAM that can be allocated to the task system. Some portion of DRAM may be unavailable due to space required to support Level-D tasks or tasks specific to alternate modes. The percentage of DRAM reserved is selected from Category 5 in Tbl. IV. For example, if 40% is selected, then under non-interleaved schemes, the DRAM consumption of generated Level-A and -B tasks on Bank 3 must be at most 40% of the 8,000 pages (refer to the discussion at the beginning of Sec. V) allocated to Levels A and B on Bank 3.

[14]Our MILP techniques are described at length in prior work [8, 20]. We refer the reader to these sources for additional information. The modifications we have made in this work are minor. As discussed in these earlier publications, the MILPs of interest take little time to solve.