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

Namhoo 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 MC² 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 MC².

1 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 MC² (mixed-criticality on multicore) [7, 8, 13, 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.

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 MC², 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 consider 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 MC². 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 MC² that is an intermediate between the extremes of replicating libraries on

---

*Work supported by NSF grants CPS 1239135, CNS 1409175, and CPS 1446631, AFOSR grant FA9550-14-1-0161, ARO grant W911NF-14-1-0499, and funding from General Motors.
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 re-mapping task page allocations in the OS. We describe our strategy for doing this in detail later.

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. 2), describe the OS modifications required to implement our library-sharing approach (Sec. 3), consider the impacts of viewing memory as a constrained resource (Sec. 4), present the above-mentioned schedulability study (Sec. 5), discuss related work (Sec. 6), and conclude (Sec. 7).

2 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, where task $\tau_i$’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 $\tau_i$ is given by $u_i = C_i/T_i$ and the total system utilization by $\sum u_i$. If a job of $\tau_i$ 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.

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[^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. Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each core. 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. 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;
MC² with hardware management. In recent work [20], we developed a new MC² 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 MC² 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 MC² 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 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 MC² implementation just described does not provide management for L1 caches, translation looksaside 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 program libraries to the MC² variant just described. A program library contains compiled functions and statically generated data used by other independently compiled programs. The classic example is the collection of C libraries that provide functions such as printf() or scanf(). In Linux and other modern OSs, libraries may be either static or shared. The two forms differ primarily in the time at which using programs are bound to the library code by linking, and how physical memory is allocated to store library content.
Programs using a library are written making references to symbols such as function names (e.g., `printf`) in the library that the compiler cannot bind to an address because the function is not part of the module being compiled. *Linking* a program with a library is the process of locating (resolving) the symbol to a function name in a library and binding the program’s symbolic reference to an address in the library.

For a static library, the references are bound explicitly as a linking step following compilation. Furthermore, this linking selects the necessary functions and data from the library and statically combines them with the program to create a complete executable program file. On systems such as Linux, this means that page frames in physical memory are needed for both the program and the library content. If the same library (e.g., the C library) is used by many or all programs, many copies of the library are necessarily duplicated and consume significant amounts of memory.

With shared libraries, linking can be postponed until the program is loaded for execution by a task (Linux process), at which time references in the program can be dynamically linked to symbols in a shared instance of the library. On Linux, this means that the shared library will be allocated space in the task’s virtual address space but the virtual addresses are mapped to a single set of page frames in physical memory that is shared by all tasks using the library. When a shared library is used by many tasks, it is quite likely that page frames will be needed for the entire library. However, since the page frames are shared among all using tasks, the net effect is often a substantial savings over static libraries.

Tasks using static libraries usually have a larger memory footprint in aggregate than do tasks using dynamic libraries. As an example, consider Tbl. 1, which gives memory-map allocations reported by *pmap* with static and dynamic libraries for the *ferret* program from PARSEC benchmark suite [5]. The first column shows the start address of mapping. The second and third columns show the size of the allocated virtual and physical memory (RSS: resident set size) 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”) allocated memory, or [stack] for the program stack.

In our previous work [7, 20], memory was assumed to be an unconstrained resource when checking schedulability. Clearly, however, memory is a constrained resource. Moreover, in application domains such as avionics, it may be further constrained by the need to support multiple modes of operation. Current aircraft can have over a dozen modes, and envisioned unmanned aircraft could have many more. As a simple example, consider a system with 20 modes and 25 distinct tasks per mode.4 Assuming the allocations shown in Tbl. 1 for each task, the total memory usage would be 1103.52MB (2260KB · 20) with static libraries and

<table>
<thead>
<tr>
<th>Address</th>
<th>KB</th>
<th>RSS</th>
<th>Mode</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>00008000</td>
<td>136</td>
<td>136</td>
<td>r-x</td>
<td>ferret</td>
</tr>
<tr>
<td>00031000</td>
<td>4</td>
<td>4</td>
<td>r--</td>
<td>ferret</td>
</tr>
<tr>
<td>00032000</td>
<td>4</td>
<td>4</td>
<td>rw-</td>
<td>ferret</td>
</tr>
<tr>
<td>00033000</td>
<td>488</td>
<td>488</td>
<td>rw-</td>
<td>[anon]</td>
</tr>
<tr>
<td>76b49000</td>
<td>520</td>
<td>520</td>
<td>rw-</td>
<td>[anon]</td>
</tr>
<tr>
<td>76bc0b00</td>
<td>872</td>
<td>872</td>
<td>r-x</td>
<td>libc-2.19.so*</td>
</tr>
<tr>
<td>76cc0c00</td>
<td>8</td>
<td>8</td>
<td>r--</td>
<td>libc-2.19.so*</td>
</tr>
<tr>
<td>76cfc00e</td>
<td>4</td>
<td>4</td>
<td>rw-</td>
<td>libc-2.19.so*</td>
</tr>
<tr>
<td>76ca00e0</td>
<td>12</td>
<td>12</td>
<td>rw-</td>
<td>[anon]</td>
</tr>
<tr>
<td>76cb2000</td>
<td>100</td>
<td>100</td>
<td>r-x</td>
<td>libgcc_s.so.1v</td>
</tr>
<tr>
<td>76cd2000</td>
<td>4</td>
<td>4</td>
<td>rw-</td>
<td>libgcc_s.so.1v</td>
</tr>
<tr>
<td>76cd3000</td>
<td>396</td>
<td>396</td>
<td>r-x</td>
<td>libm-2.19.so*</td>
</tr>
<tr>
<td>76dd4000</td>
<td>4</td>
<td>4</td>
<td>r--</td>
<td>libm-2.19.so*</td>
</tr>
<tr>
<td>76de5000</td>
<td>4</td>
<td>4</td>
<td>rw-</td>
<td>libm-2.19.so*</td>
</tr>
<tr>
<td>76df9000</td>
<td>290</td>
<td>290</td>
<td>r-x</td>
<td>libgplcblas.so.0.0.0.0.0.0</td>
</tr>
<tr>
<td>76e90000</td>
<td>4</td>
<td>4</td>
<td>r--</td>
<td>libgplcblas.so.0.0.0.0.0.0</td>
</tr>
<tr>
<td>76f2b000</td>
<td>4</td>
<td>4</td>
<td>rw-</td>
<td>libgplcblas.so.0.0.0.0.0.0</td>
</tr>
<tr>
<td>76f3d000</td>
<td>1196</td>
<td>1196</td>
<td>r-x</td>
<td>libgsl.so.0.17.0</td>
</tr>
<tr>
<td>76f3f1000</td>
<td>8</td>
<td>8</td>
<td>r--</td>
<td>libgsl.so.0.17.0</td>
</tr>
<tr>
<td>76f3f3000</td>
<td>56</td>
<td>56</td>
<td>rw-</td>
<td>libgsl.so.0.17.0</td>
</tr>
<tr>
<td>76f41000</td>
<td>92</td>
<td>92</td>
<td>r-x</td>
<td>ld-2.19.so*</td>
</tr>
<tr>
<td>76f59000</td>
<td>28</td>
<td>28</td>
<td>rw-</td>
<td>[anon]</td>
</tr>
<tr>
<td>76f60000</td>
<td>4</td>
<td>4</td>
<td>r--</td>
<td>ld-2.19.so*</td>
</tr>
<tr>
<td>76f61000</td>
<td>4</td>
<td>4</td>
<td>rw-</td>
<td>ld-2.19.so*</td>
</tr>
<tr>
<td>76ecc000</td>
<td>620</td>
<td>620</td>
<td>rw-</td>
<td>[stack]</td>
</tr>
<tr>
<td>76eec000</td>
<td>4</td>
<td>4</td>
<td>r-x</td>
<td>[anon]</td>
</tr>
<tr>
<td>fff00000</td>
<td>4</td>
<td>4</td>
<td>r-x</td>
<td>[anon]</td>
</tr>
<tr>
<td>total</td>
<td>5236</td>
<td>4868</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* This mapping can be shared by several tasks.

(a) Static linking.

Table 1: Task memory map with (a) static vs. (b) dynamic linking.

700.63MB (1428KB · 20 + 3440KB) with dynamic libraries. Thus, static linking is not tenable for this system on the machine in Fig. 3, which has only 1GB of DRAM.

3 Implementation

Static linking results in non-shared libraries. In this paper, we propose an alternative strategy, which we refer to as selective sharing. (Informally, we select “what” with “whom.”) In this section, we provide an overview of this strategy, briefly describe necessary background on Linux virtual memory and memory management, and then give a summary of the implementation mechanisms. For space reasons, the implementation description provides an outline of the approach but omits several low-level details. The complete implementation code is available online [23].

Replicating libraries. In the allocation scheme shown in Fig. 4, MC-relevant hardware isolation is provided by giving certain groups of tasks and the OS dedicated partitions in the LLC and DRAM banks. Selective sharing preserves this notion of isolation while more efficiently utilizing memory by introducing per-partition library replicas. Specifically,

---

4This is a simple example used to illustrate a point. In reality, different modes may have tasks in common and the OS also consumes DRAM space.
up to six replicas are created per library: one copy for each of the four CPUs that is shared by all Level-A and -B tasks running on that CPU, one copy shared by Level-C tasks, and one copy used by non-real-time background services.

The memory allocated to create a replica is taken from the region of DRAM used by the tasks that share it. Thus, the criticality-aware hardware-isolation properties of MC² are maintained even though libraries can be shared. In this work, we replicate only the instruction pages of a shared library, which have read/execute permissions, so we do not encounter coherence problems caused by writes.

**Virtual and physical address spaces in Linux.** Since MC² is implemented as a LITMUSH² plugin, the basic address-space concepts and lower-level memory-management mechanisms are inherited from Linux. We provide here a brief overview of those aspects of Linux that are needed to understand our implementation of selective sharing.

Fig. 5 illustrates data structures used to implement a task (Linux process) address space and manage its memory. The full virtual address space used by a task is managed by a mm_struct structure, the memory descriptor, linked from the process descriptor. Each address space is composed of a number of page-aligned memory regions in virtual memory, each represented by a vm_area_struct structure. A virtual memory region may represent an executable program, the initialized or uninitialized global variables for the program, a shared library, or dynamically growing regions such as the heap and stack. In the following, we focus only on the regions for shared libraries. Not shown in Fig. 5 are the page tables for the task, which are linked to the mm_struct structure and are used by the ARM hardware to translate the task virtual addresses to physical memory addresses.

If a region represents a shared library, then the physical memory content is read by the page-fault handler from a file in external storage when its virtual addresses are referenced by the task. Thus, the vm_file pointer is non-NULL and a set of links in the data structures shown in Fig. 5 can be traversed to find all information needed about the region for the shared library and the physical memory pages used by it. The page-fault handler allocates pages in physical memory to hold the content and sets the page tables so the virtual page addresses are translated to the allocated physical page addresses by the ARM hardware. Multiple tasks can share one physical memory copy of library code when their page tables all contain the same physical page addresses corresponding to their own virtual addresses of the library.

**Initialization.** A necessary part of the initialization procedure for tasks under MC² is a system call to the Linux mlockall() function. When this call returns successfully, all virtual memory in the task is guaranteed to have been allocated physical pages in memory, and the page tables set to translate virtual addresses to the allocated physical pages and also marked as locked so they will not be re-allocated.

This initialization step includes all shared libraries used by the task. Note, however, that the fault-handling mechanism will have allocated arbitrary free physical memory pages for the first use of a given shared library by any task. Our mechanism for sharing these libraries with isolation alters the allocations made by the fault handler and updates the page tables accordingly. It is implemented as a system call invoked as part of initialization after the call to mlockall(). The steps taken by the system call are briefly sketched below.

**Identifying shared-library pages for replication.** A task’s address space has regions for the executable code and the initialized and uninitialized variables for each shared library used by it. Although library code is shared among many tasks, each task has its own copies of all per-library global and static variables. The pages that can be replicated, the code pages of shared libraries, must satisfy the following conditions: (i) the page contains only instructions, (ii) the page is shared, and (iii) the page contents are stored in a file.

These conditions can be checked as follows. A page contains only instructions if it has read and execute permissions (in vm_area_struct in Fig. 5). A page is shared if it has a reference count exceeding two (in struct page); in contrast, a non-shared (i.e., private) page has a reference count of two. Shared libraries are loaded from external storage, so when the vm_file pointer is non-NULL in vm_area_struct, the page content is loaded from a file. Since each shared-library page used by the task had an arbitrary page allocated by the fault handler invoked by mlockall(), it is necessary to find all the page-table entries (PTEs) for shared-library pages so they can be modified to reference a replica in the appropriate DRAM region. This can be accomplished by taking virtual addresses in the range defined by the beginning and ending region addresses for a shared library in vm_area_struct and emulating in software the table walking for address translation done by the hardware to find PTEs.

**Maintaining lists of shared-page replicas.** It is not necessary to maintain a per-library list of shared pages because the first allocation during initialization using mlockall()
gives a unique set of pages for each shared library. We keep a “master” list of these initial pages and lists of the pages that are replicas of those initially allocated. During the scan of a shared library’s PTEs to locate its pages (described above), if the page does not exist in the master list, a new entry is inserted. The first time a shared library is used by a real-time task, a set of replica pages is created by allocating new pages from the appropriate DRAM region, and the content of the initial pages is copied into them. On subsequent use of the same library by another task sharing the same DRAM region, the existing replica pages are used with no further copying.

**Page remapping.** In this step, we replace the originally allocated page with a replica by altering the task’s page tables to reference the replica pages instead of the original ones. The original page, no longer referenced by the task’s page tables, is not de-allocated as it is one of the master-list pages.

As an alternative to the implementation of selective sharing just described, we also devised a second method that eliminates the need for complicated OS modifications. This second method is described in Appendix A.

**Memory footprint statistics.** To assess the impacts of selective sharing on DRAM use, we analyzed the DRAM consumption of publicly available benchmark programs in the Data Intensive Systems (DIS) stressmark suite [25], and the PARSEC benchmark suite [5]. From memory mappings, we determined the number of private pages used by several benchmark programs when libraries were non-shared (i.e., linked statically) and selectively shared. Tbl. 2 presents the results. DRAM consumption for private pages is, on average, 48% less under selective sharing compared to non-sharing.

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 issue that arose when introducing the effects of memory-capacity limits on schedulability, as necessitated by this study

### 4 Impact of Introducing Memory Constraints

Understanding the impacts of different methods for supporting shared libraries in MC² 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 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. 6 (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. 6(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. 6 (b) depicts the resulting DRAM allocations and Fig. 7 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. 6(b), the OS is no longer restricted to access only DRAM banks shared with Level C. In particular, the OS claims the first several hundred pages of physical memory on startup. 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. 7), 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Non-shared</th>
<th>Selectively Shared</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>fluidanimate</td>
<td>392</td>
<td>278</td>
<td>27</td>
</tr>
<tr>
<td>forrest</td>
<td>152</td>
<td>105</td>
<td>31</td>
</tr>
<tr>
<td>matrix</td>
<td>250</td>
<td>173</td>
<td>27</td>
</tr>
<tr>
<td>matrix</td>
<td>250</td>
<td>173</td>
<td>27</td>
</tr>
<tr>
<td>matrix</td>
<td>250</td>
<td>173</td>
<td>27</td>
</tr>
<tr>
<td>matrix</td>
<td>250</td>
<td>173</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 2: DRAM consumption for private pages of benchmark programs assuming non-shared and selectively shared libraries.
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. 5.

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. With respect to schedulability, however, there is one impact: the available DRAM capacity for Level C is reduced. Further reductions in capacity might occur at all levels, due to the need to support multiple modes, as discussed at the end of Sec. 2. Because of these sources of DRAM capacity reduction, in the experiments presented in Sec. 5, we consider scenarios in which the full capacity of DRAM might not be available.

With respect to mode changes, Fig. 6 (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.6

5 Evaluation
To assess the efficacy of selective sharing, we evaluated the schedulability of millions of randomly generated task systems under the MC2 variants listed in Tbl. 4. In denoting these variants, the prefix “NI” (resp., “II”) denotes that the interleaved (resp., non-interleaved) memory-allocation scheme was used, as depicted in Fig. 6 (b) and Fig. 7 (resp., Fig. 6 (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 MC2 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.

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 MC2 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,6

6Issues 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.
as described in Sec. 3.7.

**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. 4. 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. 4.

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

7To 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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Choice</th>
<th>Level A</th>
<th>Level B</th>
<th>Level C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: Criticality Utilization Ratios</td>
<td>A-Heavy</td>
<td>(50, 70)</td>
<td>(10, 30)</td>
<td>(10, 30)</td>
</tr>
<tr>
<td></td>
<td>B-Heavy</td>
<td>(10, 30)</td>
<td>(50, 70)</td>
<td>(10, 30)</td>
</tr>
<tr>
<td></td>
<td>C-Heavy</td>
<td>(10, 30)</td>
<td>(10, 30)</td>
<td>(50, 70)</td>
</tr>
<tr>
<td></td>
<td>All-Med.</td>
<td>(35, 45)</td>
<td>(35, 45)</td>
<td>(10, 30)</td>
</tr>
<tr>
<td></td>
<td>All-Low.</td>
<td>(35, 45)</td>
<td>(35, 45)</td>
<td>(35, 45)</td>
</tr>
<tr>
<td></td>
<td>All-High.</td>
<td>(35, 45)</td>
<td>(35, 45)</td>
<td>(35, 45)</td>
</tr>
</tbody>
</table>

| 2: Period (ms) | Short | (3, 6) | (4, 12) | (5, 25) |
|                | 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 Utilization | Light | [0.01, 0.4] | [0.04, 1.0] | [0.4, 1.0] |
|                   | Heavy | [0.25, 0.5] | [0.25, 0.5] | [0.25, 0.5] |

| 5: % DRAM Reserved for Task Set | Small | 0 |
|                                   | Moderate | 40 |
|                                   | Large | 80 |

| 6: Priv.-Page Count (SSH) in Hundreds | Light | [0.5, 2] | [0.5, 2] | [1.45, 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 3: Considered MC² variants.

**Step 3:** Based on the generated U-EDF PETs, generate PETs for the MC² 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.

**Step 5:** For each task, assign linked libraries from the list in Tbl. 5. These are libraries used by the benchmark programs listed in Tbl. 2. The first four libraries, which are shared by all benchmark programs under MC², 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 Appendix B. (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 MC² scheme in Tbl. 3. This step is also described in detail in Appendix B.

The distributions in Tbl. 4 were defined to enable the systematic study of different factors impacting schedulability, such as MC analysis, shared-library usage, and DRAM constraints. Moreover, the distribution choices in Tbl. 4 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, (C-Heavy, Long, Heavy, Light) denotes using the C-Heavy, Long, Heavy, etc., distribution choices in Tbl. 4. 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. 8. The full set of plots is available in Appendix C.
Each schedulability plot corresponds to a single scenario. To understand how to interpret these plots, consider Fig. 8(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 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. 8.

**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. 8. Under the “STC” schemes, DRAM is more of a limiting resource (since static linking wastes space), so the DRAM loss illustrated Fig. 6 (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. 8 (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. 8. (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.

### 6 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, 14, 18, 32], DRAM controllers [4, 15, 16, 22], and bus-access control [1, 2, 9, 11, 12, 27]. Other work has focused on reducing shared-resource interference when per-core scratchpad memories are used [28], accurately predicting DRAM access delays [17], throttling lower-criticality tasks’ memory accesses [34], and allocating memory [33].

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 target MC systems [4, 9, 11, 12, 15, 16, 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-related papers by our group [7, 13, 20, 24, 31], including one [7] that dealt with shared data buffers. One of these papers [20] was reviewed in detail in Sec. 2; we refer the reader to [20] for an overview of the remaining three [13, 24, 31].

### 7 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 MC² 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 MC² 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. 4. 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 MC² 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.

References

A Alternative Implementation for Selective Sharing

As an alternative to replicating libraries by performing extensive kernel modifications, we also developed a second userspace approach. This second alternative presents a trade-off that allows some complexity to be removed from the kernel in exchange for greater complexity in the user-level setup. Our evaluation in Sec. 5 was performed entirely using the previously described kernel implementation, but the method described in this appendix may be more appropriate when kernel modifications are not practical (e.g., when using a pre-certified OS that would be expensive to re-certify). For general use, however, this new method provides less transparency and requires additional setup steps. Additionally, it is possible that some executables make assumptions about their own format that are violated by the transformation described below.

Our alternative solution for replicating shared libraries consists of two steps. First, create a copy of each needed shared-library replica on disk. Second, once tasks have been allocated using our optimization approach [8], modify each task’s executable to use the correct replicas. The first of these two steps, creating library replicas, is simple as long as additional disk space is available. The second step, directing executables to use different copies of shared libraries, is non-trivial, as it requires either recompiling each executable with different dependencies or modifying the compiled versions to replace shared-library names.

We implemented the second of these possibilities by creating a tool that is capable of replacing the names of shared-library dependencies in compiled Executable and Linkable Format (ELF) executables. ELF executables contain shared-library names in sections designated as string tables, which contain a list of plain-text characters separated by null bytes. The base offset for ELF string tables is contained in the file header, and individual string references are always relative to this base offset.

Our process for replacing shared libraries begins by first finding the string table containing shared-library names, and then replacing the entries in that table. This is complicated by the fact that ELF files refer to strings by offsets, not indices, into string tables, meaning that lengthening a string will invalidate the offsets of all subsequent strings. We work around this complication by appending our string replacements to the end of the table, so only the modified strings’ offsets are affected. We next locate the ELF section containing references to shared-library dependencies, and update all string offsets to their new locations at the end of the string table. Finally, since this approach causes the string tables themselves to expand, we append a new copy of the entire string table to the end of the ELF file, and update the ELF headers to change the table’s base offset.

B 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. 5.

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. 2 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. 2), 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. 2 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 MC² schemes, we adapted the optimization scheme presented by us previously [8, 19, 20], which involves solving a mixed-integer linear programs (MILP), to determine LLC allocations while ensuring that all MC²
schedulability conditions (defined in [24]) are met. 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. 4.

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. 4. 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. 5) allocated to Levels A and B on Bank 3.

---

8Our MILP techniques are described at length in several previous publications [8, 19, 20]. We refer the reader to these sources for additional information. The modifications we have made in this work are very minor. As discussed in these earlier publications, the MILPs of interest take very little time to solve.
C Schedulability Results

AB-Mod., L., L., H., M., H., H.

A-Heavy, S., L., H., S., H., L.

AC-Mod., S., H., M., H., L.

A-Heavy, S., L., H., M., H., L.

C-Heavy, S., H., L., L., L., H.

B-Heavy, L., L., H., S., H., L.

AB-Mod., S., H., H., M., L., H.

A-Heavy, S., H., L., S., L., H.
BC-Mod., L., H., M., L., H.

All-Mod., L., H., L., S., H., L.

C-Heavy, L., H., M., L., H.

A-Heavy, L., L., H., L., H.

B-Heavy, L., L., M., H., H.

A-Heavy, S., H., L., M., L.

C-Heavy, S., L., L., L., L., L.

AB-Mod., L., L., L., H., L.
AB-Mod., S., L., S., H., H.

AB-Mod., L., L., M., L., H.

AC-Mod., S., H., M., L., H.

BC-Mod., S., H., M., H., H.

BC-Mod., L., L., M., L., L.

AB-Mod., S., L., M., L., H.

A-Heavy, S., L., M., H., H.

C-Heavy, L., L., M., L., H.
AC-Mod., S., L., L., S., L., L.

All-Mod., L., H., L., H., L.

C-Heavy, S., H., S., L., H.

BC-Mod., S., L., S., L., H.

All-Mod., L., H., S., S., L., H.

A-Heavy, L., H., H., L., H.

All-Mod., S., H., H., M., H., H.

All-Mod., S., H., L., L., H., L.
A-Heavy, S., H., L., M., L., H.

AC-Mod., S., H., L., L., H., L.

AB-Mod., L., L., M., L., L.

AB-Mod., L., L., H., M., L., L.

A-Heavy, L., L., H., S., H., H.

B-Heavy, S., L., L., L., L., L.
<table>
<thead>
<tr>
<th>Original System Utilization</th>
<th>Schedulability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>U-EDF</td>
</tr>
<tr>
<td></td>
<td>NI-STC</td>
</tr>
<tr>
<td></td>
<td>NI-SSH</td>
</tr>
<tr>
<td></td>
<td>NI-IDL</td>
</tr>
<tr>
<td></td>
<td>I-STC</td>
</tr>
<tr>
<td></td>
<td>I-SSH</td>
</tr>
<tr>
<td></td>
<td>I-IDL</td>
</tr>
</tbody>
</table>

A-Heavy, L., H., L., M., H., H.

B-Heavy, S., H., L., S., L., L.

BC-Mod., L., L., L., M., L., H.

B-Heavy, S., H., L., L., H., L.

C-Heavy, L., L., L., L., H., H.

B-Heavy, S., H., S., L., L., L.

AC-Mod., S., L., H., M., H., H.

B-Heavy, S., H., L., L., H., L.

BC-Mod., S., L., L., S., L., L.
<table>
<thead>
<tr>
<th>System Utilization</th>
<th>Schedulability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original System Utilization</td>
<td>U-EDF, NI-STC, NI-SSH, NI-IDL, I-STC, I-SSH, I-IDL</td>
</tr>
</tbody>
</table>

### Original System Utilization

- **AC-Mod.** S., L., H., S., H., L.
- **A-Heavy.** S., L., H., L., H., L.
- **C-Heavy.** S., H., M., H., L.
- **C-Heavy.** S., H., H., M., H., L.
- **C-Heavy.** L., L., H., S., H., L.
- **AB-Mod.** S., H., L., L., L., H.
- **BC-Mod.** S., H., H., S., H., H.

---

55
Original System Utilization

Schedulability

U-EDF
NI-STC
NI-SSH
NI-IDL
I-STC
I-SSH
I-IDL

All-Mod., S., H., L., L., L.

AC-Mod., S., H., H., S., H., L.

AB-Mod., L., H., M., H., H.

AC-Mod., L., L., L., L., H.

AC-Mod., L., L., S., L., H.

All-Mod., L., L., H., M., H., H.

AC-Mod., S., H., H., S., H., L.

B-Heavy, L., L., H., M., L., H.
<table>
<thead>
<tr>
<th>System Utilization</th>
<th>Schedulability</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>U-EDF</td>
</tr>
<tr>
<td>0.2</td>
<td>NI-STC</td>
</tr>
<tr>
<td>0.4</td>
<td>NI-SSH</td>
</tr>
<tr>
<td>0.6</td>
<td>NI-IDL</td>
</tr>
<tr>
<td>0.8</td>
<td>I-STC</td>
</tr>
<tr>
<td>1.0</td>
<td>I-SSH</td>
</tr>
<tr>
<td></td>
<td>I-IDL</td>
</tr>
</tbody>
</table>

- **A-Heavy, S., L., L., H., H.**
- **AB-Mod., L., H., M., L., H.**
- **AB-Mod., S., H., L., H., H.**
- **C-Heavy, S., H., L., M., L.**
- **All-Mod., L., L., H., S., H., L.**
- **All-Mod., L., L., H., L., H., H.**
- **B-Heavy, L., H., L., L., L., L.**
AB-Mod., S., H., S., H., L.

AC-Mod., L., H., L., L., H.

B-Heavy, L., L., H., M., L., L.

BC-Mod., L., H., H., S., H., L.

AB-Mod., S., H., L., S., L., H.

AB-Mod., L., L., H., L., H., L.

AB-Mod., L., L., L., L., L.

All-Mod., L., H., H., H., H., H.
AB-Mod., S., L., L., L., H.

C-Heavy, L., L., L., L., L.

AC-Mod., L., L., H., M., L.

B-Heavy, L., L., L., L., H.

AB-Mod., L., L., L., M., L.

A-Heavy, S., H., L., L., L.

BC-Mod., S., H., L., M., H.

AC-Mod., L., H., L., L., L.