Making Shared Caches More Predictable on Multicore Platforms *

Christopher J. Kenna, Jonathan L. Herman, Bryan C. Ward, and James H. Anderson
Department of Computer Science, University of North Carolina at Chapel Hill

Abstract

In safety critical domains, the usage of multicore platforms has been hampered by problems due to interactions across cores through shared hardware. The inability to precisely characterize such interactions can lead to pessimism in worst-case execution time analysis that is so great, the extra processing capacity of additional cores is entirely negated. In this paper, a new framework called MAN^{RT} *is proposed for dealing with such interactions in the context of shared caches. The major thesis of this paper is that the management of cache lines is a* synchronization problem. MAN^{RT} *controls cache usage by using recently developed optimal real-time multiprocessor locking protocols in conjunction with page coloring. The idea is to associate a set of colors with each task and require it to lock its needed colors whenever it is invoked. Experiments are presented herein that show that* MAN^{RT} *can greatly lessen observed worst-case execution times and correspondingly improve schedulability.*

1 Introduction

Multicore platforms offer the potential of enabling computationally intensive workloads in a variety of settings, with less size, weight, and power consumption. Such settings range from hand-held and embedded devices, to laptop and desktop systems, to the world's fastest supercomputers. In all of these settings, the computational capabilities enabled by multicore chips are being leveraged to realize a wealth of new products and services across many application domains. One domain, however, stands out as being largely unaffected: *safety-critical* real-time embedded systems.

In such systems, failures may have catastrophic consequences, such as loss of life or serious financial repercussions. Because of the high cost of failure, safety-critical systems must be *certified* (often by governmental or international bodies) before being deployed. Certification can be both expensive and time-consuming. Thus, it is imperative that safety-critical systems be built using hardware platforms and design processes that are certification-friendly. One of the most important tenets in this regard is that computations should be *predictable*. Predictability ensures that behaviors arising during certification reflect those that will be seen in the deployed system. Predictability is also fundamental when establishing real-time correctness.

The importance of predictability in certification explains why multicore platforms are not in widespread use in safety-critical domains. In such platforms, different cores share hardware components such as caches and memory controllers. Using current technology, very pessimistic assumptions must be made regarding the utilization of these shared resources during certification. The processing capacity lost to such pessimism can easily negate the impact of any additional cores. The resulting state of affairs is unsettling: the multicore revolution is enabling dramatically better functionality and services in many domains, but safety-critical real-time embedded systems are excluded. Unless the "predictability problem" associated with multicore platforms is addressed, functional advances in such systems will continue to be impeded. In this paper, we present an approach for addressing this problem as it applies to shared caches.

Proposed cache management approach. The "predictability problem" exists in current multicore systems because shared hardware resources are not managed in any predictable way. It has been long known that more "overt" shared resources such as shared data structures and I/O devices must be predictably managed in real-time systems using real-time locking protocols. The major thesis of this paper is that "less overt" shared hardware resources like memory controllers and cache lines require similar management.

In the case of shared caches (our focus), we show that the needed predictability can be realized by coupling recent research on optimal real-time multiprocessor locking protocols [2, 3, 4, 34] with page coloring. The idea behind page coloring is to associate "colors" with pages of physical memory according to the set of cache lines to which their contents map. If two pages have different colors, then references to memory locations within them cannot cause cache interference. We propose to view colors as shared resources that are managed using a real-time locking protocol.

Contributions. In safety-critical domains such as avionics, the multicore "predictability problem" is currently dealt with by turning off all but one core if highly-critical system components exist. Clearly, a more intelligent approach for dealing with this problem would be desirable. In this

*Work supported by NSF grants CNS 1016954 and CNS 1115284; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033.

paper, we present and evaluate a real-time cache management framework called $\$\mathsf{MAN}^{\mathsf{RT}}$ (cache management for real-time systems) that addresses this problem. The design of $\$\mathsf{MAN}^{\mathsf{RT}}$ is unique in that it reflects the viewpoint that the problem to be solved as a *synchronization problem* (as opposed to a scheduling problem). This synchronization problem is a multi-resource problem that until very recently had no known efficient solution. However, as we show, recent work on fine-grained nested real-time locking protocols can be exploited to provide such a solution [34].

Our specific contributions are as follows. First, we explain how real-time locking protocols can be applied to a page coloring scheme to manage cache lines. Second, we discuss the design of the $\$\mathsf{MAN}^{\mathsf{RT}}$ framework, which utilizes such an approach. Third, we present an experimental evaluation of $\$\mathsf{MAN}^{\mathsf{RT}}$ that examines impacts on observed worst-case execution times (WCETs). Our data shows that when $\$\mathsf{MAN}^{\mathsf{RT}}$ is applied, WCETs are greatly reduced. Nonetheless, there is a tradeoff: better WCETs are enabled at the expense of sometimes requiring tasks to block (to acquire their needed locks). In the second part of our evaluation, we examine this tradeoff. The research in this paper opens up many avenues for further work. As a final contribution, we discuss several such avenues in some detail.

Organization. In the rest of this paper, we provide needed background (Sec. 2), describe our proposed color management scheme and discuss related work (Sec. 3), present an experimental evaluation of it (Sec. 4), discuss broader challenges and future work (Sec. 5), and conclude (Sec. 6).

2 Background

In this section, we provide background on real-time scheduling and synchronization that is relevant to our work.

Task model. For simplicity, we limit attention to the well-studied implicit-deadline *periodic task model* [19] in which there are n tasks $T_1, \ldots, T_n$ to be scheduled on m processors.[1] Each task T_i is characterized by a *worst-case execution time* (*WCET*) e_i, *period* (separation time between invocation, or *jobs*) p_i, and relative deadline $d_i = p_i$. T_i's needed processing capacity is given by its *utilization*, $u_i = e_i/p_i$.

Scheduling algorithms. Variations of $\$\mathsf{MAN}^{\mathsf{RT}}$ can be applied under any global, clustered, or partitioned job-level static priority (JLSP) scheduler, assuming either hard real-time (HRT) (all deadlines must be met) or soft real-time (SRT) (deadlines may be missed but tardiness must be bounded) correctness. (We assume familiarity with these terms—recall that global and partitioned scheduling are special cases of clustered scheduling.) All JLSP schedulers that we specifically consider use either *earliest-deadline-first* (EDF) or *static* (task) *priority* (SP) prioritizations. We use the prefixes "C-," "P-," and "G-," to indicate clustered, partitioned, and global scheduling, respectively; for example, G-EDF denotes global EDF scheduling. While $\$\mathsf{MAN}^{\mathsf{RT}}$ is not tied to a particular scheduling approach, the implementation presented in Sec. 3 assumes a P-SP-scheduled HRT system.

Multiprocessor real-time synchronization. The design of $\$\mathsf{MAN}^{\mathsf{RT}}$ leverages recent work on asymptotically optimal multiprocessor real-time locking protocols. (We assume familiarity with locking-related terms like "critical section" as well.) In the protocols we consider, a task waits for a lock by *suspending* execution. Locking protocols must ensure that *priority inversion blocking* (*pi-blocking*) can be analytically bounded. Pi-blocking is the duration of time a job is blocked while a lower-priority job is running.

A thorough understanding of how multiprocessor real-time locking protocols work is not required for our purposes. However, it is instructive to understand the basic synchronization problem that must be solved. As described more fully in Sec. 3, under page coloring, physical pages of memory are assigned colors in a way that ensures that accesses within differently colored pages cannot cause cache conflicts. Moreover, accesses within pages that are colored the same cause conflicts only if the number of "ways" in the cache is exhausted. We view each color as a shared resource that has a number of "replicas" given by the number of cache ways. Each task may require locks on several color replicas before it can execute. This means that we need a synchronization protocol that can be used to manage a set of resources, where each resource has multiple replicas, and tasks may need to acquire locks on several replicas simultaneously. Fortunately, the needed locking functionality is provided by recent work on optimal real-time multiprocessor locking protocols that allow *nested* lock requests [34]. Actually, we utilize improved variants of these protocols that support *dynamic group locks* on multi-replica resources [33]. Dynamic group locks provide functionality similar to nested locks except that multiple lock requests by a task can be satisfied simultaneously instead of individually.

3 Proposed Cache Management Scheme

In this section, we describe the design of $\$\mathsf{MAN}^{\mathsf{RT}}$, using the machine used in the experiments presented in Sec. 4 to illustrate important ideas.

3.1 System Background

System description. The machine we consider is a single-socket Intel Core i7 920 platform with four 2.66 GHz cores running Linux. We assume the shared cache at the lowest (third) level, the *L3 cache* (hereafter shortened to L3) is to

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

be managed. The 8 MB L3 is shared by all four cores and is *unified*, *i.e.*, it stores both instructions and data. Each core also has private (non-shared) L1 instruction and data caches (32 KB each) and a unified, private L2 cache (256 KB).

System memory is byte-addressable. Memory addresses are categorized as either linear or physical.[2] In our system, a *linear address* (also called a *virtual address*) refers to a memory location in a flat, unsegmented address space over all possible byte addresses. A *physical address* is used to address physical memory (RAM). If paging (see below) is disabled, a linear address is used as a physical address without modification; otherwise, an additional translation step is needed.

Paging. Under paging, the linear address space is divided into *pages* that are then mapped to physical memory. Paging is used to support *virtual memory*, which makes the mapping of linear addresses to physical addresses transparent to tasks and isolates the linear address spaces of different tasks. Additional protection mechanisms to isolate user- and kernel-level memory are also supported. We assume that paging is used to support virtual memory and isolation, but *demand paging* (*i.e.*, the dynamic swapping of pages to/from disk) is not used since the resulting disk I/O can cause unpredictable delays. User-level programs use 4 KB pages, while the kernel uses a larger page size for its virtual memory.[3] Fig. 1 depicts the translation of linear to physical addresses using 4 KB pages. As explained in the figure's caption, the translation is carried out via a hierarchy of *paging structures*. Each task has such a paging hierarchy, since it has its own linear address space.

Caches. We describe how physical memory is cached by focusing on the L3 of our platform. Memory is transferred into the cache in 64-byte blocks called *lines* as illustrated in Fig. 2. The L3 is a 16-way *set associative* cache. Such a cache is partitioned into multiple sets of 16 "slots" each, where each slot or "way" can hold one line. A given memory line may be stored in any of 16 slots in the set to which that memory line address maps. Since our cache has 16 ways and is 8 MB in size with a 64 B line size, there are $(8\,\text{MB}/16\,\text{ways}) \times (1\,\text{way}/64\,\text{B}) = 2^{13}$ sets in this cache. These sets are represented by rows in the figure. Since a cache line is 64 B, the offset of a byte within a cache line is given by the least significant six bits in the physical memory address. The next 13 bits of the address provide an index into one of the $2^{13} = 8,192$ sets in the cache. The remaining bits of the address are used as a tag, which is stored in a "directory" entry to disambiguate the lines within a set. Other levels of cache operate similarly to the L3. However, they use a different number of bits in the index and tag due

[2] *Logical addresses*, which are used in a segmented memory model, also exist, but the Linux kernel utilizes segmented memory in a very limited way. Therefore, we do not consider logical addressing further.

[3] The platform supports additional user-level page sizes, but we do not consider them here.

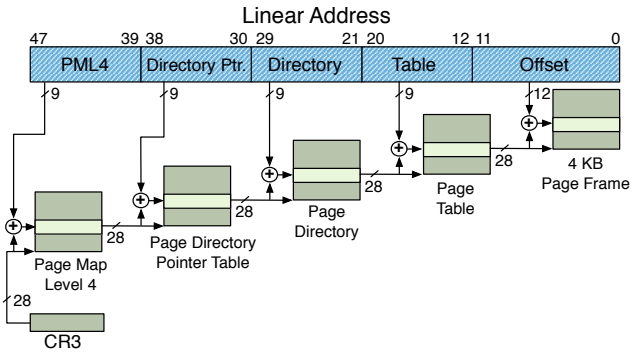


Figure 1: A 48-bit linear memory address (hatched, in blue) is translated to a 40-bit physical memory address. The root paging structure, the PML4 table, is located via the processor's CR3 register. Address translation is an iterative process where each set of nine bits in the range 47:12 of the linear address indexes into a paging structure, which contains the physical memory address of the next paging structure in the translation process or the desired page frame. The least significant 12 bits of the linear address index into the page frame to select a byte.

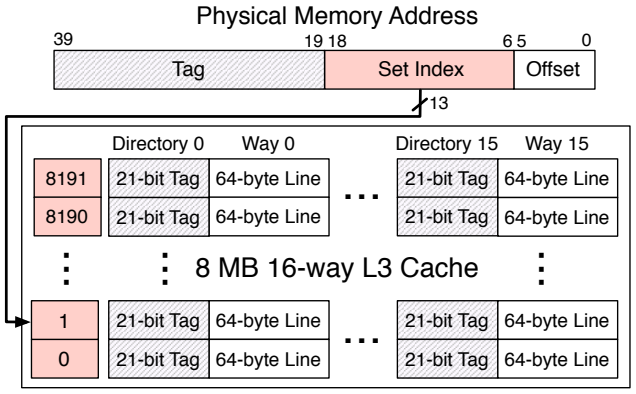


Figure 2: L3 cache structure.

to differences in cache size and associativity. Also, the L1 cache is virtually indexed, while the L2 and L3 caches are physically indexed. All caches are physically tagged.

3.2 MAN^{RT} Design

We now present a software-based approach for reducing conflict misses in shared caches, which occur when an excessive number of lines must be cached in the same set. This approach is based on an existing idea, called *page coloring*, which attempts to optimize the usage of physically-indexed, set-associative caches with a cache size large enough such that the bits determining the set index extend past the bits determining the page frame offset (like our L3 cache). Our contribution is the addition of a real-time synchronization protocol that allows the number of ways in a cache to be predictably allocated. We begin with a brief overview of page coloring, based upon the hardware platform described

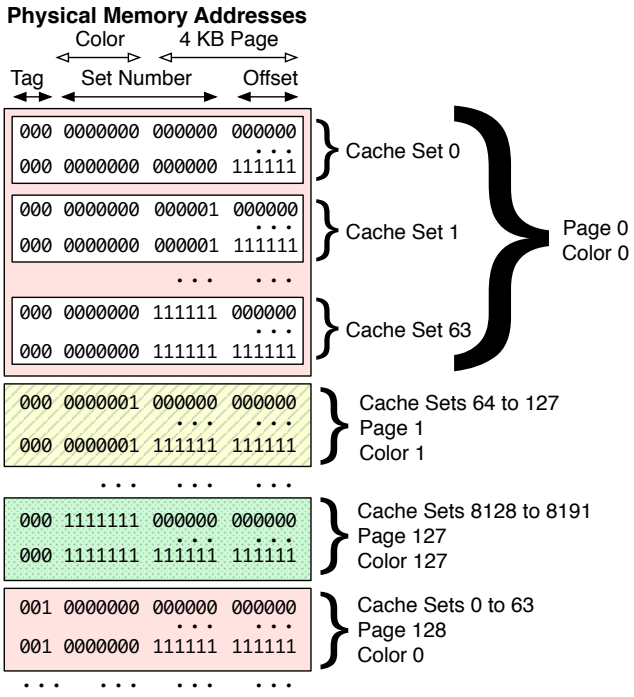


Figure 3: Figure showing how colored pages with specific addresses max to cache sets.

above.

Page coloring. Under page coloring, pages of physical memory are assigned "colors" in a way that ensures that bytes within differently-colored pages can never map to the same cache sets. The basic idea is simple. Assign the color 0 to Page 0 in physical memory; correspondingly, color all cache sets to which words in Page 0 can map with the color 0. Continue this process by considering Page 1, then Page 2, *etc.*, "wrapping" (re-using previous colors) when necessary.

Fig. 3 illustrates this idea as applied to the L3 of our example system. The top of the figure depicts how physical addresses are used by both the L3 and the OS. The cache (black arrowheads) uses the least significant six bits of the physical address to locate a byte in a cache line. The next 13 bits of the physical address are used to select a cache set, and the remaining bits are used as a tag (which is shortened in the figure for brevity). The OS (white arrowheads) uses the least significant twelve bits to locate a byte in a 4 KB frame, and the next seven bits as a "color." Since pages are allocated to applications in units of $4\,\mathrm{KB} = 2^{12}\,\mathrm{B}$, there are $(2^{12}\,\mathrm{B/page}) \times (1\,\mathrm{line}/64\,\mathrm{B}) = 64$ cache lines per page. Each line in the page has a unique cache-set index; thus, the first 64 cache sets are colored the same as Page 0, the next 64 the same as Page 1, and so on. Since seven bits are used to determine colors, Page 128 maps to the same cache sets as Page 0, and thus is colored the same way.

Cache lines as shared resources. We propose using a multiprocessor real-time synchronization protocol to enable tasks to "lock" their needed colors (*i.e.* the colors of the pages they will access) prior to execution. With a 16-way set-associative cache as in our example, there are 16 "replicas" of each color available. That is, tasks may lock up to 16 pages of the same color concurrently without causing cache conflicts.

The synchronization problem that must be solved is as follows: we have a set of shared resources (the available colors), each with multiple replicas (16 in our example), and prior to execution, each job must first acquire (or lock) a specified number of replicas of a set of specified resources (corresponding to the coloring of its pages and how many pages it must access).[4] This is a multi-replica, multi-resource synchronization problem where tasks may hold locks on multiple replicas simultaneously. As noted earlier in Sec. 2, protocols for supporting such locks that have optimal pi-blocking factors can be obtained from prior work [33, 34].

3.3 Related Work

Cache miss avoidance is important for predictability and for increasing processing efficiency. Thus, unsurprisingly, methods similar to our proposed approach have been studied in the past, and it is important to differentiate between them. An approach called *cache locking* (not to be confused with our use of term "locking") has been proposed wherein designated cache lines are "locked down" in the cache so that they cannot be evicted [6]. Similarly, an approach called *cache partitioning* has been proposed that attempts to mitigate the impact of cache conflicts by allocating sections (or partitions) of the cache to specific tasks. In [14], several cache partitioning algorithms for uniprocessor systems are reviewed. Cache partitioning can be done automatically by the compiler [21], but the source code of programs must be available for compilation and large portions of memory must be allocated as padding to achieve the desired code and data placement. To remedy this, partitioning at the OS level was proposed [17]. This approach can be applied dynamically, transparently, and without access to the source code of the running application program, since it relies on the paging component of the OS. However, it may be difficult to size partitions so that the cache is efficiently utilized from a system-wide perspective.

Cache partitioning is actually a special case of our approach in which inter-task cache conflicts are entirely eliminated, and hence the synchronization component of $\$\mathsf{MAN}^{\mathsf{RT}}$ is obviated. However, with our generalization, a system designer may freely slice the cache into (potentially overlapping) areas and still ensure that executing jobs will not suffer from cache conflicts due to other currently ex-

[4] If job execution times are lengthy, then it is possible to break jobs into sub-jobs and require sub-jobs to acquire and release locks.

ecuting jobs. Other benefits arise in various extensions to the basic design of $\$\mathsf{MAN}^{\mathsf{RT}}$ that is our focus here, such as the ability to "overload" certain colors in a mixed-criticality setting (see Sec. 5).

A system execution model called PREM [25] has been proposed that takes an approach similar to ours but in a more scheduling-oriented way. Specifically, PREM uses scheduling to reduce or eliminate contention for shared resource accesses, including main memory. However, unlike $\$\mathsf{MAN}^{\mathsf{RT}}$, PREM is restricted to single-core systems.

In work on timing analysis tools, analysis methods pertaining to memory hierarchies have been proposed, ranging from fundamental static cache analysis for first-level caches [10, 13, 16, 23, 27, 32] to multi-level ones [9, 15, 22] and multicores [15, 36]. At the hardware level, *cache bypass* [8] reduces cache conflicts by caching only memory blocks known to be reused; this requires special hardware instructions, while page coloring is transparent to the application and compilation process. Compiler techniques have been proposed to support single-task page swapping points for the demand paging of instructions based on page coloring [26]; in contrast, our proposed work focuses on multi-task cache conflicts remedied by page coloring. Much work has focused on hardware support to make multicore platforms more predictable at the processor [24] and network-on-chip (NoC) interconnect levels [1, 7, 28, 30]. In contrast, we propose a unique software-only approach applicable to commodity hardware without modifications.

4 Evaluation

As explained earlier, $\$\mathsf{MAN}^{\mathsf{RT}}$ lessens WCETs at the expense of potentially blocking jobs due to color acquisition. In this section, we experimentally evaluate this tradeoff.

4.1 Measurements

We begin by considering an experiment conducted to assess the impact of $\$\mathsf{MAN}^{\mathsf{RT}}$ on WCETs.

Experimental setup. We implemented a prototype of $\$\mathsf{MAN}^{\mathsf{RT}}$ within LITMUS$^{\mathrm{RT}}$, a Linux-based multiprocessor real-time OS [18]. As safety-critical real-time systems are our main focus, we implemented a variant of $\$\mathsf{MAN}^{\mathsf{RT}}$ that uses partitioned rate-monotonic (P-RM) scheduling and considered only harmonic periodic task systems. The protocol used to implement color locking under partitioned scheduling uses a mechanism called "priority boosting," which effectively makes scheduling nonpreemptive [33, 34]. We assume that tasks are independent (no shared resources other than cache lines), all task pages are memory-resident, and tasks do not share pages. The machine that we used in our experiments is the same as that described in Sec. 3. All page coloring was done with respect to the last level of cache (LLC) on this machine, which is its L3.

As discussed in Sec. 3, our test platform supports 128 page colors. We used a custom `mmap` memory allocation function in order to modify the page tables of the backing user process for each task to map pages with its assigned colors into its virtual address space. As explained in Sec. 5, we leave the full exploration of color assignment heuristics to future work. For the purposes of this paper, we tested several heuristics and used one that spreads a task's pages throughout the LLC, as it produced the best performance under all schedulers of the heuristics we tested. In this initial prototype, we only implemented coloring with respect to tasks' data pages. We leave the coloring of other areas of memory (such as pages for task and kernel code, stacks, shared libraries if binaries are not statically compiled, *etc.*) as future work. We believe this is reasonable, as each of our implemented tasks has a small per-job code footprint (200 B) that does not make any system calls, and operates only on the colored memory it allocates.

Task set generation. Harmonic task sets were generated by selecting periods uniformly from $\{25, 50, 100, 200\}$ and utilizations uniformly from $[0.01, 0.05]$ for the number of tasks desired. Tasks were assigned to processors using the worst-fit heuristic. Any task set that could not be so assigned was discarded. Each task set that could be assigned was scaled to within 1% of its breakdown utilization so that it was difficult to schedule. This was done to ensure that the tested task sets put enough pressure on the LLC to note differences among different allocation schemes.

The lowest-priority task on each CPU was used to provision a periodic server for best-effort work instead of executing as a periodic task. The server allows best-effort work to participate in $\$\mathsf{MAN}^{\mathsf{RT}}$, thereby isolating best-effort work the same way $\$\mathsf{MAN}^{\mathsf{RT}}$ isolates real-time tasks from one another. Each of the m servers dequeued aperiodic jobs from a global FIFO queue. We ran eight aperiodic job generators that generated aperiodic jobs with an exponentially distributed execution cost with mean 3 ms truncated to within $[2, 100]$ ms and inter-job arrival times exponentially distributed with mean 100 ms truncated to 200 ms.

We defined each task via a code sequence that requires each of its jobs to read or write one or more elements from a cache line in an array of memory according to some configurable parameters (described below). We determined the number of elements to access per-job by converting from the task's specified execution time using access rates obtained for each array size in an unloaded system. Our test platform has a cache prefetcher that cannot be disabled. To ensure that it does not activate and decrease any observed WCET, we configured each task to access the elements in its given array in a random order. We utilized performance counters in test runs to ensure that the prefetcher did not activate.

Task parameters. In our experiments, we considered the following task parameters: *cache footprint*, or simply *footprint*, F, *working set size* (*WSS*) W, *number of tasks*, and

read-write ratio. The footprint is the size of the array above, in bytes. W indicates the size of a *job's* cache footprint: the array elements a job randomly accesses are confined to a region of size W within the larger (task) footprint of size F. The read-write ratio is the number of cache line reads per cache line write for each task. In preliminary experiments we conducted, the WSS and footprint parameters were found to affect observed WCETs significantly, while the read-write ratio had little affect. For this reason, only one read-write ratio choice is considered in the experiments below.

System overheads. Before presenting the results of our WCET experiments, we briefly comment on system overheads. $\$MAN^{RT}$ adds some additional complexity to the OS's scheduling logic. We found that worst-case scheduling overheads increased from 7–10 μs to 15–20 μs, while average overheads increased from 3 μs to 6 μs. Such increases are fairly negligible in comparison to task execution times.

WCET Experiments. In order to evaluate the coloring and synchronization aspects of $\$MAN^{RT}$ separately, we considered the following three system configurations:

C1: $\$MAN^{RT}$ with page coloring as described above.

C2: Page coloring as in (C1) is used but $\$MAN^{RT}$ is not.

C3: Neither $\$MAN^{RT}$ nor page coloring is used.

Under C3, we purposely allocated pages to reflect a worst-case scenario with high memory contention by pathologically requiring all tasks to share the same color. Such a scenario actually has practical relevance. Many safety-critical systems also have security requirements. Thus, it is desirable to prevent malicious tasks from attempting to purposefully evict the cache lines of another task, thereby inflating its WCET.

In the experiments below, we compare observed WCETs under these three configurations. While it might be preferable to conduct such a comparison based on WCETs predicted by timing analysis tools, adequate tools for multicore do not yet exist. Also, note that observed WCETs lower bound predicted ones (if the prediction is safe). Thus, C1–C3 give some indication of how a tool might perform given varying degrees of information about cross-core cache interactions (virtually no information in C3). Note that one advantage of $\$MAN^{RT}$ is that it enables uniprocessor cache-related timing analysis results to be applied to shared multicore caches. Thus, given the simplified structure of our task system code, we speculate that observed WCETs better approximate what a realistic tool could produce under C1 than under C2 and C3.

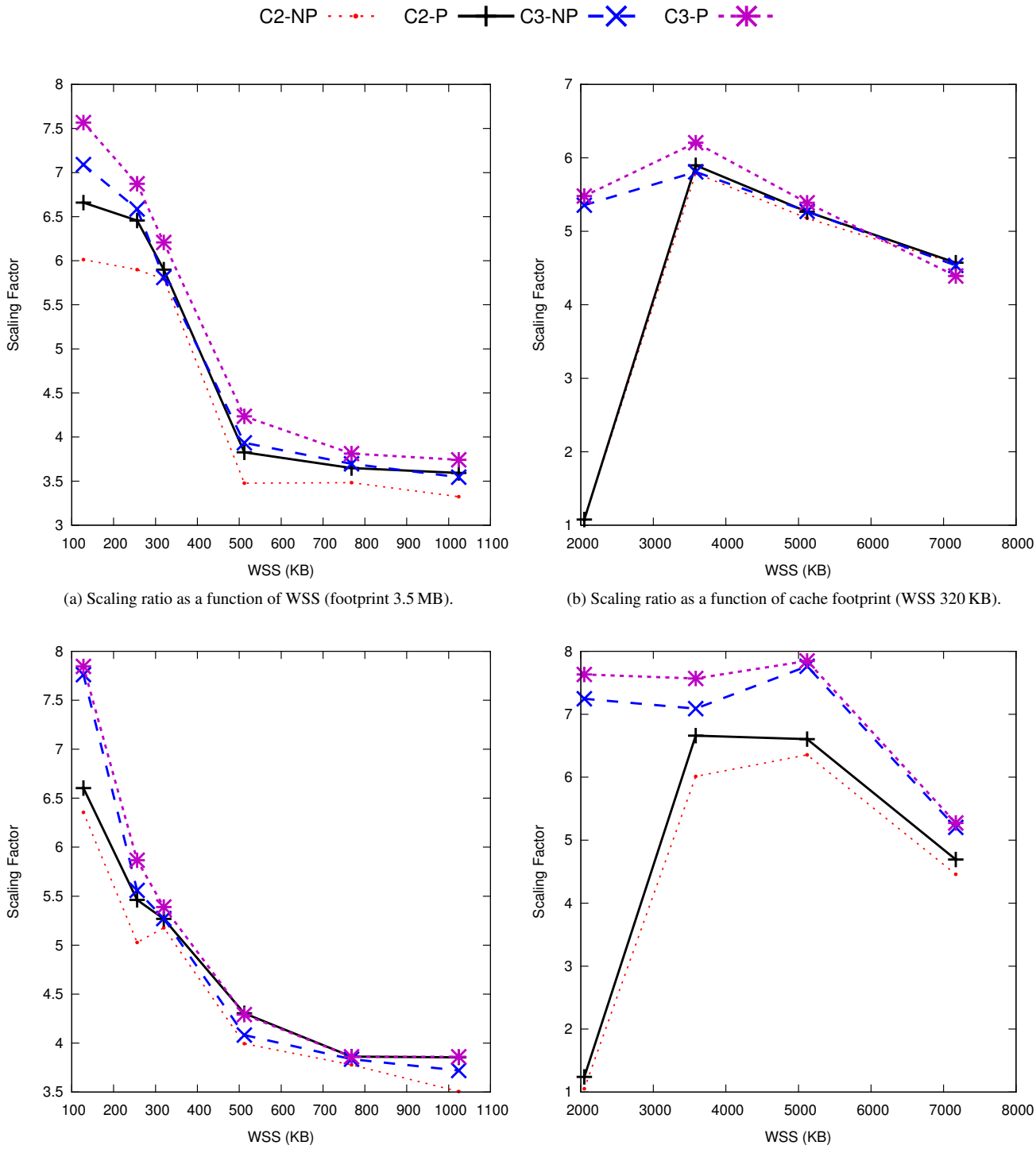
Recall from earlier in this section that when $\$MAN^{RT}$ is used, P-RM schedules tasks nonpreemptively. To assess the impact of nonpreemptive execution under the configurations that do not require it, we considered two variants of each of C2 and C3, one with preemptive P-RM and a second with nonpreemptive P-RM. We denote these configurations as C2-P, C2-NP, C3-P, and C3-NP.

We executed 36 task systems (several times each, with different random number generator seeds) under each of the five configurations. In each of run, we recorded each task's largest observed WCET. We ran each task system for 10 seconds, as WCETs were seen to converge by that amount of time.

Results. We compare WCETs under C2-P, C2-NP, C3-P, and C3-NP to those under C1 by reporting *scaling factors* of the form x/y, where x is a given task's WCET under one of the former configurations, and y is its WCET under C1. Fig. 4 gives worst-case scaling factors (the largest observed for any task) for various parameter settings. Such ratios are shown as a function of WSS in insets (a) and (c) and footprint in insets (b) and (d). These graphs are representative of others that we must omit due to space constraints.

Several interesting observations can be made from the data in Fig. 4. First, the ratios for C2 and C3 under nonpreemptive scheduling are smaller than under preemptive scheduling. This is because preemptions cause greater cache misses due to affinity loss. Second, the ratios for C2 are lower than for C3, but typically not by much. C3 represents a very pathological situation that should give high ratios. However, C2's comparable ratios indicate that coloring alone cannot alleviate all cache conflicts. Third, as seen in insets (a) and (c), all ratios decrease as WSS increases. This is due to cache line reuse. A job with a fixed execution time but a smaller WSS iterates over its entire WSS more than the same job with a larger WSS. Fourth, $\$MAN^{RT}$ generally results in much better WCETs. As an exception, in insets (b) and (d), page coloring alone (C2) does almost as well for a footprint of around 2 MB. This is because the combined footprint of four executing jobs in this case fits within the LLC. However, with larger footprints, the capacity of the LLC is exhausted, and performance under cache partitioning degrades. Note that, in insets (b) and (d), the ratios get a bit smaller for very large footprints. This is because, as the cache footprint increases, the number of sets from which a job might draw its WSS increases. This results in a greater number of colors that a given WSS may use, and reduces the probability of cache conflicts. Still, for these very large footprints, the depicted ratios indicate inferior performance relative to $\$MAN^{RT}$.

To confirm that our test machine it not atypical, we ran a simple benchmark to measure the LLC-to-memory latency ratio [20, 29] on several machines in our lab. On our test platform, this ratio is approximately four (60 ns to memory versus 15 ns to LLC). We obtained similar ratios for other machines. While a ratio of four is smaller than what is seen in Fig. 4, this is because the benchmark tool runs a single task in isolation, while the task systems used to obtain Fig. 4 often fully saturate the memory subsystem.



(a) Scaling ratio as a function of WSS (footprint 3.5 MB).

(b) Scaling ratio as a function of cache footprint (WSS 320 KB).

(c) Scaling ratio as a function of WSS (footprint 5 MB).

(d) Scaling ratio as a function of cache footprint (WSS 128 KB).

Figure 4: Scaling factors for configurations C2-P, C3-P, C2-NP, and C3-NP as a function of WSS (insets (a) and (c)) and cache footprint (insets (b) and (d)). A ratio exceeding one indicates larger observed WCETs than $\$MAN^{RT}$.

Observed schedulability. The observed WCET results presented above indicate that schedulability is likely improved under $\$MAN^{RT}$. To see if this is so, we ran additional experiments in which schedulability metrics were assessed. In these experiments, we executed task systems in a similar manner as before, but with WSSs varying from 64 KB to 1024 KB, and a 3.5 MB footprint. System utilizations were scaled up to within 5% of system breakdown utilization, to sufficiently stress the system while providing some flexibility for runtime effects. For each task in these systems, we calculated the *deadline miss ratio* (the ratio of jobs that failed to complete by their deadline) and the *relative tardiness* (the average tardiness of each task divided by its period). Averages for these metrics are plotted in insets (a) and (b) of Fig. 5 as a function of WSS for our five tested configurations.

As can be seen, $\$MAN^{RT}$ (configuration C1) misses the fewest deadlines and experiences the lowest relative tardiness of all configurations. This indicates that the extra synchronization-related blocking under $\$MAN^{RT}$ is offset in most case by improvements in WCETs. Note that C2-NP (coloring and non-preemptivity) sometimes exhibits similar values for these metrics. However, as remarked earlier, we expect that if we were using predicted WCETs from timing analysis tools, then there would be a larger difference between the WCET values arising under $\$MAN^{RT}$ and the other configurations. This might more clearly separate $\$MAN^{RT}$ from C2-NP when examining the metrics presented in Fig. 5. In any event, this figure clearly illustrates the value of more predictable shared cache management from a schedulability perspective.

5 Discussion

In this section, we discuss various challenges we encountered while implementing $\$MAN^{RT}$ and how to relax some of the assumptions we made in Sec. 4.1.

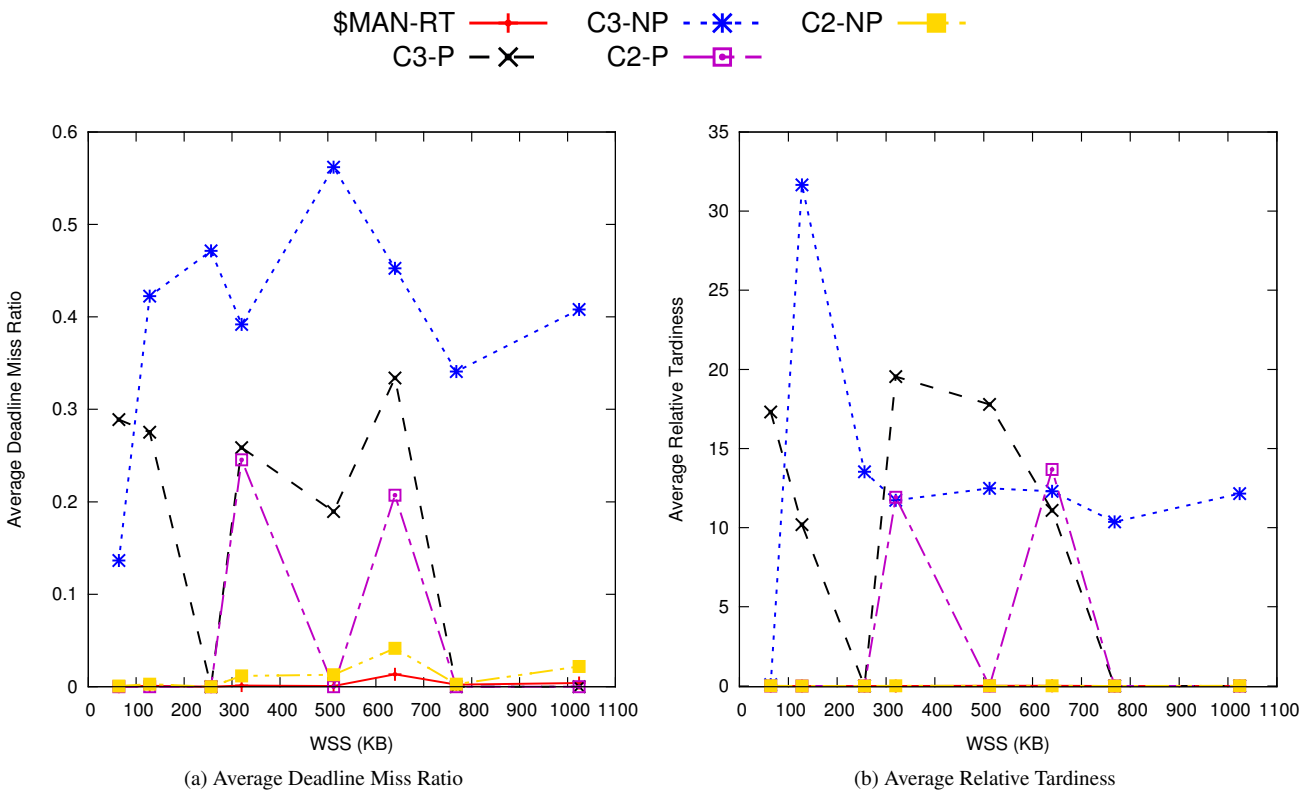
Index hashing. As explained in Sec. 3, 13 bits of a physical address are used as a base-2 index into our system's set of cache lines in the LLC. Consequently, adjacent cache-line-sized blocks of memory map to adjacent lines in the cache. In order to distribute cache lines uniformly across the cache, manufacturers may opt to use *index hashing*, where a portion of a physical address is hashed and the result is used to index the cache sets. Sun opted to include this feature in the UltraSPARC T2 processor after noticing certain overutilized LLC cache sets called "hot-spots" occurring at particular address offsets in the T1 processor; however, this feature can be optionally disabled [31, 35]. Hybrid approaches are also possible; *e.g.*, the Intel "Sandy Bridge" microarchitecture uses physical-address hashing to determine which *slice* of the LLC a physical address maps to; each core controls an equally sized *slice* of the cache [11].

The hash functions used by closed-source processors are often confidential, so it is impossible to know which cache set memory will map to based on its physical address (short of attempting to reverse-engineer the processor). In addition, there exist known hash functions that use address bits lower than the page size as input, which results in OS pages that cannot be colored in a straightforward manner, if at all. Either of these situations precludes $\$MAN^{RT}$. Fortunately, index hashing seems to be isolated to desktop- and server-level systems at present. When precise cache control and predictability are of paramount importance, as they are in real-time systems, performance improvement heuristics such as index hashing can be detrimental—caches that employ index hashing can exhibit pathological, worst-case behavior [12]. Manufactures should give end-users the option of turning off index hashing.

Color assignment. The assignment of colors to tasks (under consideration of blocking bounds) is a complex problem that warrants further investigation. We note that the problem of assigning tasks to cache partitions is NP-hard [5], so we did not attempt to solve the similar problem of coloring task pages optimally for the purposes of this paper. In the version of this problem considered herein, it is assumed that the number of replicas of a color that exist in the locking protocol is equal to the number of cache ways. However, in some situations it might make sense to relax this assumption. For example, one could imagine a mixed-criticality system where "low-criticality" colors are allowed to be over-utilized in the worst case, *i.e.*, low-criticality tasks are permitted to incur a limited number of cache conflicts while executing.

Synchronization. The initial prototype system described in this paper was developed assuming that tasks are independent and do not access shared pages. In future work, we intend to eliminate these restrictions. Of particular interest is the interplay between synchronization protocols used to protect ordinary shared resources and those used to manage cache lines. Although we have effectively treated entire jobs as critical sections in this paper, in reality these are not "real" critical sections, so the OS could force a lock release on a job at any time. We intend to investigate the impact this has on schedulability. Color locks can be allocated to sub-jobs rather than full jobs to lessen lock holding times. We have explored this possibility to a limited extent, but job splitting strategies certainly warrant further investigation. At the other extreme, it might be desirable for certain critical tasks to retain color locks across several job releases, under certain system modes. We intend to investigate this as well.

Other issues. We would like to perform additional studies like the one in this paper in which scheduling other than P-RM is assumed, and in which the coloring of all pages (not just task data pages) is considered. We would also like to



(a) Average Deadline Miss Ratio

(b) Average Relative Tardiness

Figure 5: Average deadline miss ratio and average relative tardiness observed under each experimental configuration.

perform similar studies in which WCET values from timing analysis tools are considered instead of observed WCETs.

6 Conclusion

In this paper, we have described a shared cache management framework called $\$MAN^{RT}$, the design of which is based on the premise that shared cache lines are resources that require synchronized access. We also presented experimental results obtained from an initial prototype of $\$MAN^{RT}$ on a quad-core machine. These experiments indicate that synchronized cache management can lessen WCETs, make them more predictable, and positively impact schedulability. This work motivates many future research directions, which we have discussed as well.

References

[1] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable SDRAM memory controller. In *CODES+ISSS '07*, 2007.

[2] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2011.

[3] B. Brandenburg and J. Anderson. Optimality results for multiprocessor real-time locking. In *RTSS '10*, 2010.

[4] B. Brandenburg and J. Anderson. Real-time resource-sharing under clustered scheduling: Mutex, reader-writer, and k-exclusion locks. In *EMSOFT '11*, 2011.

[5] B. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA '08*, 2008.

[6] M. Campoy, A.P. Ivars, and J.V.B. Mataix. Static use of locking caches in multitask preemptive real-time systems. In *IEEE/IEE Real-Time Embedded Sys. Workshop*, 2001.

[7] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22:414–421, 2005.

[8] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS '09*, 2009.

[9] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS '08*, 2008.

[10] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache

performance. *IEEE Trans. on Comp.*, 48(1):53–70, 1999.

[11] Intel. *Intel 64 and IA-32 architectures optimization reference manual*, 2012.

[12] M. Kharbutli, Y. Solihin, and J. Lee. Eliminating conflict misses using prime number-based cache indexing. *IEEE Trans. Comput.*, 54(5):573–586, 2005.

[13] S. Kim and S. Min. Efficient worst case timing analysis of data caching. In *RTAS '96*, 1996.

[14] D. Kirk. SMART (strategic memory allocation for real-time) cache design. In *RTSS '89*, pages 229–237, 1989.

[15] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In *9th Int'l Workshop on WCET Analysis*, 2009.

[16] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *RTSS '96*, 1996.

[17] J. Liedtke, H. Härtig, and M. Hohmuth. OS-controlled cache predictability for real-time systems. In *RTAS '97*, 1997.

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

[19] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *JACM*, 30:46–61, 1973.

[20] L. McVoy and C. Staelin. Lmbench – tools for performance analysis. `http://www.bitmover.com/lmbench/`.

[21] F. Mueller. Compiler support for software-based cache partitioning. In *LCTRTS '95*, 1995.

[22] F. Mueller. Timing predictions for multi-level caches. In *LCTRTS '97*, 1997.

[23] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, 2000.

[24] M. Paolieri, E. Qui nones, F. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA '09*, 2009.

[25] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *RTAS '11*, 2011.

[26] I. Puaut and D. Hardy. Predictable paging in real-time systems: a compiler approach. In *ECRTS '07*, 2007.

[27] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *RTAS '06*, 2006.

[28] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS '11*, 2011.

[29] R. Ruggiero. Measuring cache and memory latency and CPU to memory bandwidth. White paper, Intel Corporation, 2008. Available online `http://download.intel.com/design/intarch/papers/321074.pdf`.

[30] R. Stefan, A. Molnos, A. Ambrose, and K. Goossens. A TDM NoC supporting QoS, multicast, and fast connection set-up. In *DATE '12*, 2012.

[31] Sun Microsystems. *UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007*, 2007.

[32] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *RTSS '03*, 2003.

[33] B. Ward and J. Anderson. Nested multiprocessor real-time locking with improved blocking. Submitted to RTSS '12, 2012. http://www.cs.unc.edu/˜anderson/papers.html/.

[34] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *ECRTS '12*, 2012.

[35] D. Weaver. *OpenSPARC Internals*. Sun Microsystems, 2008.

[36] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS '08*, 2008.

A Schedulability Study

In this appendix, we present additional experiments that were conducted to assess the utility of \$MAN$^{\mathsf{RT}}$ from a schedulability perspective and to examine the tradeoff between improved WCETs enabled by \$MAN$^{\mathsf{RT}}$ and the increased blocking that occurs as a result. In these experiments, we generated HRT task systems to be scheduled on the four-core system described in Sec. 3, with total utilization $U \in \{0.1, 0.2, \ldots, 2.0\}$. These parameters were inspired by avionics applications in which HRT, SRT, and best-effort (BE) subsystems exist, where the HRT subsystem is relatively small.[5] Within each task system, tasks had either *light* (distributed uniformly between 0.001, and

[5] We have been told by multiple industry sources that the HRT subsystem typically represents *at most* 20% of the overall workload; on a four-core machine, this translates to a total utilization of less than 1.0.

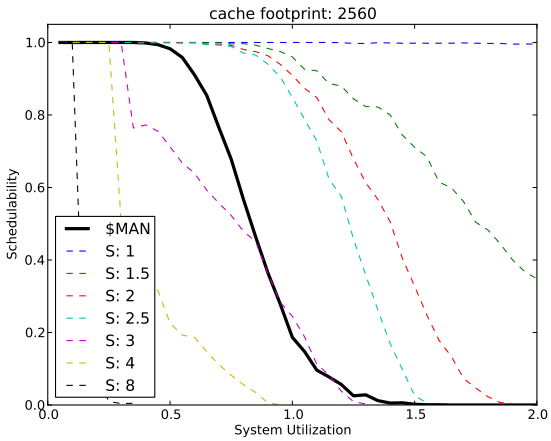


Figure 6: Schedulability of task systems with $\$\text{MAN}^{\text{RT}}$, and without $\$\text{MAN}^{\text{RT}}$ but with execution costs inflated by a factor of S.

0.1), *medium* (distributed uniformly between 0.1 and 0.4), or *heavy* (distributed uniformly between 0.5 and 0.9) utilizations. All tasks in each generated task system were assumed to have the same size cache footprint, which we varied within $\{32, 64, 128, 256, 512, 1024, 2048, 2560, 3072, 4096, 6144, 8192\}$KB. We discuss here a few selected graphs from this experimental design space to demonstrate the benefits of $\$\text{MAN}^{\text{RT}}$.

For each task system scheduled under $\$\text{MAN}^{\text{RT}}$, we assigned colors by *wrapping*, similar to how wrapping is described in Sec. 3. Replicas of colors are assigned to tasks one at a time by iterating through the set of colors, and wrapping, or reusing previous colors if need be. For each color, blocks of 4, 8, and 16 replicas are assigned at a time to an individual task. Tasks are colored in priority order, and the colors of one task start where the previous task left off until all tasks have been fully colored.

Note that for cache footprints less than $1/m$ of the cache (2048KB for our cache of size 8096KB), the tasks may be cache partitionable, while task systems with very large cache footprints, analytically, are serialized on a single processor. Fig. 6 plots schedulability—*i.e.*, the fraction of generated task systems that were deemed schedulable—as a function of total utilization for task systems in which all tasks have a footprint of 2560KB, and thus the cache is not partitionable. In this graph, the solid line shows the schedulability of task systems scheduled with $\$\text{MAN}^{\text{RT}}$, while the dashed lines show the schedulability of the same task systems after their execution costs have been inflated by a scaling factor of S. This scaling factor represents the degree of pessimism that multicore timing-analysis tools must incorporate into their analysis on account of shared caches. The experiments presented earlier in Sec. 4 suggest that scaling factors of three to eight or even greater may be seen in practice.

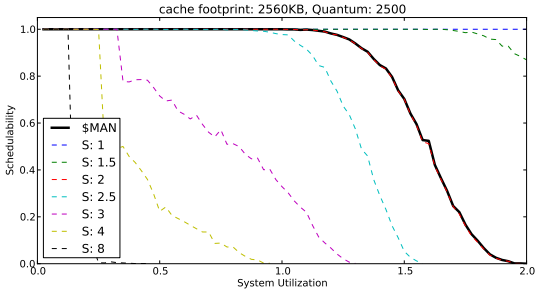


Figure 7: Schedulability of task systems with and without $\$\text{MAN}^{\text{RT}}$, with job and period slicing.

As seen in Fig. 6, the blocking caused by $\$\text{MAN}^{\text{RT}}$ can be detrimental from a schedulability perspective, even for task systems that could be scheduled on a single core. However, there are a number of techniques that can be applied to improve blocking bounds and improve schedulability. By applying *job slicing*, *i.e.*, by breaking each job into sub-jobs, we can exploit the fact that a job's execution is not a true critical section. This in affect allows lower-priority jobs to forfeit their colors to higher-priority jobs. While this will force low-priority jobs to reload their cache when they resume execution, this cost can be quantified and incorporated in schedulability analysis. Furthermore, uniprocessor timing analysis is still possible, as the cache is isolated during the duration of a sub-job.

In addition, *period slicing*, in which a job's execution is split across several smaller periods, can also be used to improve blocking bounds. If all tasks have their periods sliced such that their sub-periods are equal, and all sub-jobs release at the same time, then there can be no blocking on tasks on a local processor. We applied both job slicing with sub-jobs of length at most 2.5ms, and period slicing with sub-periods of 25ms to task systems with the same properties as those in Fig. 6. As seen in Fig. 7, these techniques can greatly reduce worst-case blocking and improve schedulability. In this figure, schedulability under $\$\text{MAN}^{\text{RT}}$ is commensurate with assuming a scaling factor of two (and no cache isolation). As noted above, the experiments presented earlier in Sec. 4 suggest that scaling factors much higher than two are likely relevant in practice. These results suggest that $\$\text{MAN}^{\text{RT}}$ can have a profound affect on the schedulability of multicore systems.

Supporting BE work. As noted earlier, in avionics systems (as well as many other applications), HRT work is often supported in conjunction with BE work. In such systems, the improved WCETs under $\$\text{MAN}^{\text{RT}}$ allow systems to complete more BE work, and do so in a more timely fashion. $\$\text{MAN}^{\text{RT}}$ allows BE work to execute when HRT work is blocked on cache locks. This can be supported by simply partitioning the HRT and BE subsystems in the cache so

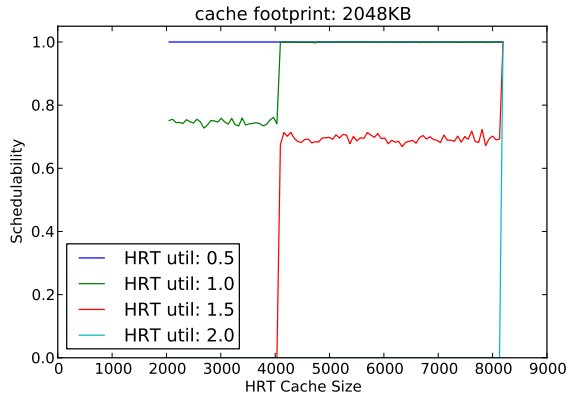


Figure 8: Schedulability of HRT components scheduled with $\$\mathsf{MAN}^{\mathsf{RT}}$ and restricted to a subset of the cache.

that the BE work does not have to contend for cache locks with the HRT work.

To maximize the performance of the BE work, it is best to have a large BE cache partition. This in turn reduces the size of the HRT partition, and increases the amount of blocking that HRT jobs experience. In our schedulability experiments, we also investigated the schedulability of task systems when the size of the HRT cache is restricted to different size cache partitions. An example graph from this study is given in Fig. 8. In this graph, we consider the schedulability of HRT subsystems (*i.e.*, HRT task systems) with total utilizations of $\{0.5, 1.0, 1.5, 2.0\}$. In these experiments, each task had a cache footprint of size 2048KB, and each generated HRT task system was restricted to $\{2048, 2064, \ldots, 8192\}$KB of the 8192KB cache. In this figure, we can see that HRT task systems with smaller utilization can be scheduled in smaller cache partitions, leaving more of the cache available for BE work. Note that, when the size of the HRT cache partition is large enough to enable cache partitioning to be applied to HRT tasks *within* that partition, there is no blocking. However, many scenarios occurred in our experiments in which HRT cache partitioning was not possible and color locks had to be utilized to obtain a schedulable system.