Making Shared Caches More Predictable on Multicore Platforms *

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

Abstract

In safety-critical cyber-physical systems, 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 worst-case execution time pessimism that is so great, the extra processing capacity of additional cores is entirely negated. In this paper, several techniques are proposed and analyzed for dealing with such interactions in the context of shared caches. These techniques are applied in a mixed-criticality scheduling framework motivated by the needs of next-generation unmanned air vehicles.

1 Introduction

Multicore platforms offer the potential of enabling computationally intensive workloads in many settings, with less size, weight, and power (SWaP) 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 of 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* cyber-physical embedded systems.

Examples of such systems include avionic and automotive systems, medical systems for diagnosis and treatment, and smart robotic systems that function in environments where failures cannot be tolerated or are difficult to correct (*e.g.*, planetary rovers). A common characteristic of these and other safety-critical systems is that 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 expensive and time-consuming. Thus, it is imperative that safety-critical systems be built using certification-friendly hardware platforms and design processes. 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.

*Work supported by NSF grants CNS 1016954, CNS 1115284, CNS 1218693 and CNS 1239135; ARO grant W911NF-09-1-0535; AFOSR grant FA9550-09-1-0549; and AFRL grant FA8750-11-1-0033. The first author was supported by an NSF graduate research fellowship.

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 cyber-physical 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 consider this problem in the context of shared caches.

Next-generation UAVs. Our work is motivated by ongoing research with colleagues at Northrop Grumman Corp. (NGC) on real-time operating system infrastructure for next-generation unmanned air vehicles (UAVs). Currently, avionics manufacturers resolve the multicore "predictability problem" by turning off all but one core if highly critical system components exist. They strongly desire a more intelligent solution for dealing with this problem.

Next-generation UAVs will have significant computational workloads (hence the need for multicore) and system components of varying criticalities [22, 29]: these range from "safety critical" tasks that perform functions required for stable flight, to less-critical dynamic planning computations that produce better results over time. Our work to date with our NGC colleagues has been directed at supporting such mixed-criticality task systems on multicore platforms. This work has resulted in the development of a multicore resource allocation framework called MC^2 (mixed-criticality on multicore) [12, 20]. Here, we consider the problem of adding proper shared cache management to MC^2.

We address this problem by considering several cache management schemes that utilize page coloring in some way. Under page coloring, pages of physical memory are assigned "colors" in a way that ensures that differently colored pages cannot cause cache conflicts. As discussed later, prior work has shown that page coloring alone can often completely eliminate inter-task cache conflicts in real-time systems. However, the problem of optimally allocating colors is NP-hard in the strong sense [5]. Also, due to constraints on memory, which can arise for a variety of reasons (*e.g.*, SWaP limitations, the need to support many pro-

cessing modes, *etc.*), conflict-free color assignments may be unobtainable. Motivated by these observations, we present a new idea to enable coloring to be more flexibly utilized. Specifically, we consider treating colors as shared resources to which accesses must be arbitrated, either by a real-time locking protocol or a scheduling algorithm.

Contributions. We consider cache management in a variant of MC^2 in which both higher-criticality (HC) hard real-time (HRT) tasks and lower-criticality (LC) soft real-time (SRT) tasks must be supported. Our objective is to enable worst-case execution time (WCET) reductions in HC tasks by properly managing shared-cache accesses; such reductions can enable more HC tasks to be supported.

We consider two basic cache-management approaches, which we term *cache locking* and *cache scheduling*. Under cache locking, portions of the cache are viewed as non-preemptive resources that are accessible via a locking protocol. Under cache scheduling, portions of the cache are viewed as preemptive resources that are "scheduled." This approach results in a scheduling problem that is quite difficult generally; however, we show that known uniprocessor schedulability analysis can be used to provide a sufficient schedulability condition that performs well in many cases. We note that in some special cases, these cache management strategies are obviated, in which case these approaches reduce to cache partitioning.

To evaluate the presented schemes, we implemented each within LITMUS$^{\mathrm{RT}}$ [18] and conducted experiments on an NVidia Tegra T30 quad-core[1] ARM Cortex A9 system to assess relevant overheads and enabled WCET improvements. We then conducted overhead-aware schedulability experiments in which these schemes were compared against each other and to the alternative of applying no cache management. Our major conclusions from these experiments are as follows. First, proper shared cache management can enable significant WCET reductions; on our test platform, observed WCETs were reduced up to almost five-fold. Second, while cache management entails some overhead, these WCET reductions enable significant schedulability gains. Third, cache scheduling generally enables higher schedulability gains than cache locking and can be more easily and flexibly applied.

Organization. After providing needed background (Sec. 2), we describe our cache management techniques (Sec. 3) and their implementation, overheads, and enabled WCET improvements (Sec. 4), present a schedulability-based evaluation of them (Sec. 5), and conclude (Sec. 6).

2 Background

In this section, we provide background on scheduling, synchronization, and page coloring that is relevant to our work.

Task model. We consider real-time workloads that can be defined using the implicit-deadline *periodic task model* and we assume familiarity with this model. We specifically consider a task system $\tau = \{T_1, \ldots, T_n\}$, which is to be scheduled on m processors,[2] where task T_i's *period* and *worst-case execution time* (*WCET*) are denoted p_i and e_i, respectively. We denote the jobs released by T_i as $J_{i,1}, J_{i,2}, \ldots$. (We sometimes omit the job index and let J_i denote an arbitrary job of T_i.) We denote T_i's *utilization* by $u_i = e_i/p_i$. Algorithms for scheduling such a task system may follow a *partitioned approach* (tasks are statically assigned to processors), a *global scheduling* approach (any task may execute on any processor), or some hybrid of the two.

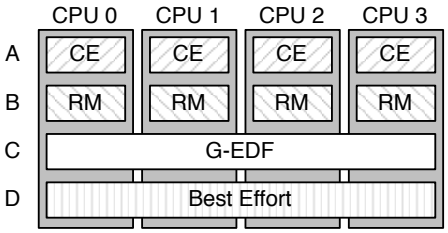


Figure 1: Scheduling under MC^2 on a four-processor system.

MC^2. We consider periodic task systems scheduled under the MC^2 mixed-criticality framework [12, 20]. Under mixed-criticality schedulability analysis [29], different methods for determining task execution times are assumed to be applied at different criticality levels, with greater pessimism at higher levels. For example, *provably correct* upper bounds on execution times from timing analysis tools might be assumed at the highest criticality level, while *observed* worst-case times from profiling might be sufficient at lower levels. When checking real-time correctness, L variants of a system with L criticality levels must be analyzed: in the level-l variant, level-l execution times are assumed for *all* tasks.

In MC^2, four criticality levels exist, denoted A (highest) through D (lowest), as shown in Fig. 1. Higher criticality tasks are statically prioritized over lower criticality ones. Level-A tasks are partitioned and scheduled on each processor using a table-driven cyclic executive. Level-B tasks are also partitioned but are scheduled using a rate-monotonic (RM) scheduler on each processor.[3] Level-A and -B tasks are required to be simply periodic (all tasks commence execution at time 0 and periods are harmonic). Level-C tasks are scheduled via a global earliest-deadline-first (G-EDF) scheduler. Level-D tasks are scheduled with no real-time guarantees on a best-effort basis (so we do not consider them further). A task's execution time at its own criticality level is treated as an operating-system (OS) enforced execution budget: if a job of a task T_i has an execution time exceeding T_i's budget, then more than one budget allocation will be required to service it. Level-A and -B tasks are HRT, while level-C tasks are SRT (under the "bounded deadline tardiness" definition of SRT [7]). Most interesting cache-related issues are exposed by focusing only on levels B and

[1] Enabling even a quad-core machine to be used in an avionics setting would be a significant innovation.

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

[3] An EDF scheduler can be optionally used at level B.

C (one HRT level and one SRT level). Thus, due to space constraints, we hereafter focus on systems in which only levels B and C are present. We note that systems with two criticality levels have been the predominate focus of prior work on mixed-criticality scheduling (see, *e.g.*, [2, 3, 8]).

Page coloring. The ARM platform used in our experiments has four cores that share an L2 cache. In this paper, we consider page coloring with respect to this cache. The L2 cache on this platform is a 1 MB 8-way set associative cache: it stores contents of physical memory in 32 B units called "lines," each line of physical memory maps to a particular cache "set," each such set can store 8 lines (equivalently, there are eight "ways" per set), and in total there are 2^{12} sets. The physical memory of this platform is subdivided into 4 KB pages. To envision the coloring process, consider each page in sequence. For the first page in memory, assign the color "0" to it, and assign the same color to the cache sets to which its contents map. Then, since each page consists of $4\,\mathrm{KB}/(32\,\mathrm{B/line}) = 128$ lines, sets $1 - 128$ are assigned color 0. Repeat this process, assigning color 1 (mapping to sets $129 - 256$) to the second page in memory, color 2 (sets $257 - 384$) to the third page, and so on. Then, after the 32^{nd} page, all 2^{12} sets will have been used and color assignments will "wrap," *i.e.*, the 33^{rd} page will map to the same cache sets as the first, so we reuse color 0 for it. Continuing this process, each page will be assigned to one of 32 colors. Moreover, two pages that are assigned different colors will map to different cache sets and thus cannot conflict with each other in the cache.

In Sec. 3, we consider techniques that exploit page coloring to eliminate or control cache conflicts. In discussing these techniques, we limit attention to non-shared task data pages, as only these pages are managed in the initial prototype system described in Sec. 4. We define the *working set size* (*WSS*) of a task to be the size (in bytes) of the set of data pages it may access in one job, *i.e.*, the size of its per-job *working set* (*WS*). We assume that each task's WSS is at most the size of the shared cache.

Multiprocessor real-time locking. Some of the cache management schemes we consider utilize multiprocessor real-time locking protocols. In the protocols we consider, tasks wait 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. Per-task bounds on pi-blocking are required when analyzing schedulability. We let b_i denote the pi-blocking bound for task T_i.

On a multiprocessor system, the actual definition of pi-blocking depends on how schedulability analysis is done [4]. For some schedulers, suspensions are notoriously difficult to analyze, so *suspension-oblivious* (*s-oblivious*) analysis is applied: jobs may suspend, but each e_i must be *analytically* inflated by b_i prior to applying a schedulability test to account for lock-related delays. We utilize s-oblivious analysis in this paper. Some of the nuances of such analysis can best be explained by comparing it to *suspension-aware* (*s-aware*) analysis, which explicitly accounts for b_i and is available for some schedulers.

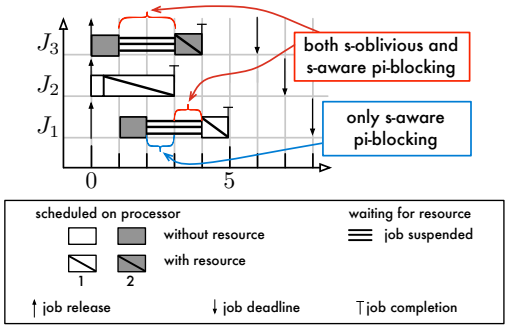


Figure 2: Example (from [4]) of s-oblivious and s-aware pi-blocking. G-EDF scheduling on two processors is assumed. J_3 suspends during $[1, 3)$, and since no higher-priority jobs exist it is pi-blocked under either definition. J_1, suspended during $[2, 4)$, suffers pi-blocking under either definition during $[3, 4)$ since it is among the m highest-priority pending jobs, but only s-aware pi-blocking during $[2, 3)$ as J_3 is pending but not ready then.

Since suspended jobs are counted as demand under s-oblivious analysis, the mere *presence* of m higher-priority jobs rules out a priority inversion, whereas only *ready* higher-priority jobs can nullify a priority inversion under s-aware analysis.[4] Accordingly, under **s-oblivious** (resp., **s-aware**) schedulability analysis, a job J_i incurs *s-oblivious* (resp., *s-aware*) *pi-blocking* at time t if J_i is pending but not scheduled and fewer than m higher-priority jobs are **pending** (resp., **ready**). This is illustrated in Fig. 2. Prior research has shown that s-aware and s-oblivious analysis are comparable in terms of schedulability achievable in practice [4].

Cache-related locking problem. We now describe the basic synchronization problem that arises when using locking protocols for cache management (protocol-specific details are discussed in Sec. 3). When using such protocols, each color is viewed as a shared resource that has a number of "replicas" as given by the number of cache ways, as illustrated in Fig. 3. Before a job commences execution, it must first lock a replica of each color that it requires (as given by the pages it will access). If the job accesses r pages with the same color, then it must lock r replicas of that color. The needed synchronization protocol must enable a set of shared resources to be managed, where each resource has multiple replicas, and jobs may need to lock several replicas simultaneously. In actuality, such a protocol is utilized by the OS when making scheduling decisions, *i.e.*, the jobs themselves do not acquire and release color-related locks. This means that the OS must know the pages a job will access prior to making a scheduling decision.

3 Cache Management Techniques

In this section, we present techniques for managing accesses to a shared cache by level-B tasks in MC^2 (recall our assumption that level A is not present). Our goal is to enable

[4] A job is *pending* if it has been released but is not completed. A pending job is *ready* if it is available for execution. A suspended job is not ready.

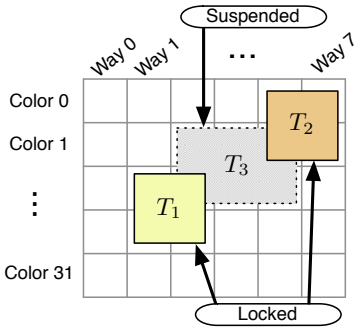


Figure 3: Color locking.

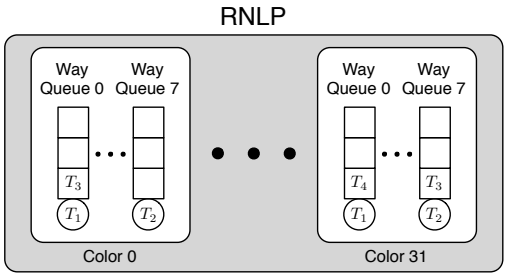


Figure 4: Illustration of the RNLP queue structure.

WCET reductions and greater predictability in such tasks. We do not apply these cache management techniques to the level-C subsystem, which is provisioned using less pessimistic bounds on WCETs (*e.g.*, the level-C subsystem may be provisioned based on observed worst- or average-case execution times), which are not as significantly impacted by the possibility of adverse interactions across shared hardware resources. Furthermore, we assume that tasks of different criticality levels are *partitioned* in the cache, *i.e.*, they are allocated such that they do not share colors (support for such allocation is described in Sec. 4). The term "task" is used to refer to a level-B task in the rest of this section.

3.1 Color Management Alternatives

We begin by broadly describing color management strategies and our assumptions. First, we assume an inclusive write-back cache for which the OS can precisely control which cache sets and ways a task's data is loaded into. Second, we assume that the OS has access to hardware mechanisms that enable ways of the cache to be *locked* (and later *unlocked*) such that data in a locked way is protected from being evicted (equivalently, the way is unavailable for allocation). Third, we assume that the physical memory pages, and therefore the color requirements of each task, have been previously assigned, and that tasks have been partitioned onto processors. Finally, we assume that the memory a task uses is preallocated before the task system begins execution, as is common in real-time systems. We discuss the realization of the first two points on our test platform in Sec. 4. In Sec. 5, we address the third requirement by describing methods to assign colors.

The effect of these assumptions is that once data is loaded into a cache way and it is locked, all subsequent memory accesses to that data will hit in the cache. With this property, uniprocessor timing analysis tools, which are much less pessimistic than multicore ones,[5] can be employed to more accurately estimate WCETs for level-B tasks. Furthermore, observed WCETs will be improved as the number of cache misses is provably reduced.

Under these assumptions the cache can be treated as either a non-preemptive or a preemptive resource. This gives rise to four different classes of allocation policies, depending upon whether the processor or the cache is preemptive. We claim, however, that either both should be preemptive, or both should be non-preemptive. If the processor is non-preemptive and the cache is preemptive, then it is possible for a job to be scheduled while its needed cache colors have been preempted (*i.e.*, are not available for it to use). Alternatively, if the cache is non-preemptive and the processor is preemptive, then colors could be locked by a job that is later preempted and thus unable to use them. We therefore consider only the non-preemptive case, which we call *cache locking*, and the preemptive case, which we call *cache scheduling*.

3.1.1 Cache Locking

Under cache locking, a job must hold a *color lock* for all of its needed colors before execution, and it does not release its color locks until the end of its execution, *i.e.*, its execution requirement is a critical section. This ensures cache isolation for each job during the entirety of its execution. This policy can be realized by using a multiprocessor real-time locking protocol to arbitrate access to colors and treating each job's execution time as a critical section.

For this purpose, we leverage the RNLP [30],[6] a recently developed multiprocessor real-time locking protocol that optimally supports the simultaneous locking of multiple resources. The RNLP controls access to all cache colors and their respective ways. For each way of each color, there is an associated FIFO-ordered *way queue* of jobs. This architecture is depicted in Fig. 4. The head of each way queue is assumed to have acquired the associated way, though it does not execute until it has acquired all needed ways. A job J_i atomically requests all colors it requires before it can commence execution. For each color c from which J_i requests r_c ways, J_i is enqueued in the shortest r_c way queues associated with c. A job in a way queue that is either waiting for a resource or scheduled with its needed ways is considered non-preemptive. Therefore, no other jobs on the processor can be either scheduled or enqueued in the way queues. Because there can be at most m jobs total in all way queues, and because jobs enqueue in the shortest way queues, the maximum duration of blocking for all cache colors is $O(mr/k)$ where k is the number of ways available

[5] Actually, research on multicore timing analysis is still in its infancy, so it is doubtful that such tools are used much (if at all) in practice. Note that, to entirely reduce timing analysis to a uniprocessor problem, cache management would need to be supported for all pages.

[6] We made a minor modification to support replicated resources.

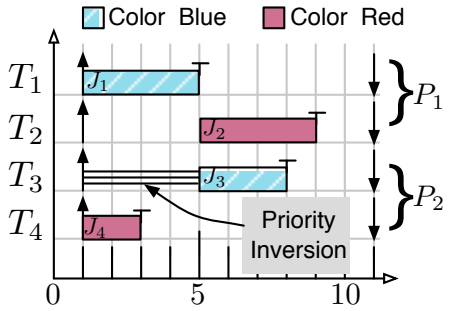


Figure 5: Example of the trade-off between priority inversion and improved WCET (explained in text).

and r is the maximum number of ways per color requested by any job. Under s-oblivious analysis, a non-preemptive job is analytically treated as scheduled, even when it is not actually scheduled. This non-preemptivity can cause $O(mr/k)$ non-preemptive blocking for other jobs. Thus, the total duration of blocking is $O(mr/k)$.[7]

Ex. 1. Consider a two-processor system, with two tasks on each processor, as depicted in Fig. 5. For simplicity, assume that the cache is direct-mapped (*i.e.*, only one way per color). Assume that jobs J_1 and J_2 (resp., J_3 and J_4) are partitioned onto processor P_1 (resp., P_2). Also, assume the lower-indexed jobs have higher priority. Let jobs J_1 and J_3 share blue (striped in the figure), and jobs J_2 and J_4 share red (solid). If all jobs are released synchronously on both processors, then without cache locking, both jobs J_1 and J_3 would be concurrently scheduled, and would conflict in the cache. However, as shown in Fig. 5, J_1 is concurrently scheduled with J_4. These jobs do not conflict in the cache. When J_1 completes, J_2 and J_3 can execute without conflicting in the cache. In this example, there is a *priority inversion* when job J_3 cannot run, and instead the lower-priority job J_4 executes. *Our goal is to show that the effect of such priority inversions is offset by the improved WCETs afforded by cache isolation*.

Period splitting. Non-preemptive processor scheduling, a byproduct of our locking protocol, can cause pi-blocking that can be detrimental to the schedulability of a task set.

Ex. 2. Consider a system where T_i has a higher priority than T_j. If $e_j \geq p_i$, then the system may be unschedulable because T_j might be non-preemptive throughout the entirety of T_i's period, causing T_j to miss its deadline.

The undesirable effects of non-preemptivity can be ameliorated by exploiting the fact that cache-related critical sections are not "true" critical sections: they can be preempted, albeit with an additional cost to reload evicted cache lines that are later required. Under s-oblivious analysis (recall from Sec. 2), non-preemptive blocking can be eliminated using a technique called *period splitting*. Under period splitting, each task's period is set to the shortest period in the system and its execution time is scaled accordingly. Thus, each job actually executes as a sequence of subjobs.

Ex. 3. Consider a system with two tasks, T_1 and T_2, where $e_1 = 1$, $p_1 = 3$, $e_2 = 6$, and $p_2 = 9$. Under period splitting, T_2's period is reduced to match p_1, without altering its utilization. Thus, after period splitting, $e_2 = 2$ and $p_2 = 3$.

Since level B in MC^2 is simply periodic, period splitting ensures that subjobs are always released in phase. Under this assumption, the RNLP guarantees that all s-oblivious pi-blocking is caused by tasks on remote processors.

Job splitting. Such remote blocking can clearly cause the system to be unschedulable. It is particularly detrimental if critical-section lengths are highly variant.

Ex. 4. Consider a system in which a color is shared between two tasks T_1 and T_2, where $e_1 = 1$ and $p_1 = e_2 = 16$. If T_1 blocks for 16 time units on T_2, then it will miss its deadline.

We can mitigate such detrimental blocking by breaking each task into multiple subtasks that are scheduled on the same processor and that all have the same period, but smaller execution times, such that all subtask utilizations sum to the original task's utilization. This is similar to inserting preemption points into jobs of a task, but is implemented by enforcing budgets. We call this technique *job splitting*. Note the difference between job and period splitting: under job splitting, tasks are broken into many subtasks with smaller utilizations, while under period splitting, a task's period is reduced, while maintaining its utilization.

Ex. 4 (cont'd). If T_2 is split into 16 subtasks, each with an execution cost of one, then worst-case blocking for T_1 is improved from 16 to one.

Job and period splitting, which can potentially be applied together, can reduce pi-blocking bounds by ameliorating the adverse effects of highly variant critical-section lengths. However, under either scheme, there is additional overhead for each subjob, such as costs due to scheduling decisions and reloading cache lines. Such overheads could be prohibitive if splitting is too "fine-grained."

3.1.2 Cache Scheduling

While non-preemptively scheduling tasks with respect to the cache maximizes reuse of cache lines within each job, it can cause adverse blocking, as we have seen. Alternatively, we can treat the cache as a *preemptive resource*, a technique we call *cache scheduling*. In this case, color replicas are preemptively "scheduled." When a task is scheduled with respect to a set of color replicas, it has exclusive access to those replicas and thus will not experience cache conflicts with tasks on remote processors. However, it may be preempted to allow a higher-priority task to access some replica. Similar to job and period slicing, such a preemption may force the preempted task to reload its WS. However, the cost of such reloads can be analyzed and incorporated into schedulability conditions using existing techniques as described in [4, Chap. 3].

Ex. 5. Consider the two-processor schedule depicted in Fig. 6, where each processor, P_1 and P_2, has two assigned

[7] More exact blocking analysis, though possible, is omitted due to space constraints.

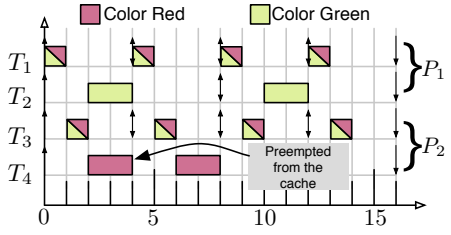


Figure 6: Preemptively scheduling the cache.

tasks with utilization $1/4$. Tasks T_1 and T_3 are defined by $(e_1, p_1) = (e_3, p_3) = (1, 4)$ and both share red and green. Task T_2 is defined by $(e_2, p_2) = (2, 8)$ and requires green only, while T_4 is defined by $(e_4, p_4) = (4, 16)$ and requires red only. RM priorities are applied to tasks on both the cache and their respective processors. Thus, tasks T_1 and T_3 are scheduled before T_2 and T_4. At time $t = 4$, T_4 is preempted by the higher-priority task T_1 on processor P_1.

We believe that by viewing cache colors as preemptive processors we expose a new scheduling problem. The problem is how to schedule and verify the schedulability of a task system in which each task T_i has an additional parameter, R_i, which gives a set of *processor requirements*. A processor requirement $(a, s) \in R_i$ specifies that for T_i to execute, it must be scheduled on a processors from the set of processors s.

Ex. 6. For a task T_i assigned to processor P_2 requiring four replicas of color blue, of which there are 16 ways, B_1, $\ldots$, B_{16}, we have $R_i = \{(1, \{P_2\}), (4, \{B_1, \ldots, B_{16}\})\}$.

This scheduling problem is a generalization of gang scheduling [13], and scheduling with processing set restrictions [16]. We leave this general problem for future work, but present more simplistic analysis by reducing the problem to a uniprocessor scheduling problem.

Reduction to uniprocessor analysis. Fundamental to our analysis is the observation that there are task systems for which the processor may not be the constrained resource—instead the cache may be overutilized, causing the task system to be unschedulable.

Ex. 7. Consider two tasks that share the same set of colors and have a total utilization greater than one. The tasks may be partitioned onto either one or two processors. In the former case, the processor is overutilized, and the task system is unschedulable. In the latter case, the tasks must run sequentially to ensure cache isolation, and thus the task system is unschedulable despite the additional processing capacity of the second processor. This is because the cache is the constrained resource, not the processors.

We next formalize the reduction to uniprocessor analysis mentioned above and demonstrate how the utilization of cache colors can be evaluated to check schedulability with respect to both cache colors and processors.

We define a binary relation for *direct contention* D, such that two tasks are related if they share colors or a processor,

$$D = \{(T_i, T_j) \in \tau^2 \mid C_i \cap C_j \neq \emptyset \vee P(T_i) = P(T_j)\},$$

where C_i is the set of colors that task T_i requires, and $P(T_i)$ is the processor on which T_i is partitioned. Let D^+ be the transitive closure of direct contention, D. We define each equivalence class in D^+ to be a logical *cache processor*. By definition, all tasks on the same physical processor must be on the same cache processor, and thus the number of cache processors is at most m.

We evaluate the schedulability of each cache processor as a uniprocessor, and apply known schedulability tests. In MC^2, which uses RM priorities on each physical processor and simply periodic periods, the schedulability test for a physical processor P_j is $\sum_{T_i \text{ on } P_j} u_i \leq 1$. We can also schedule the cache using RM priorities, and allow new jobs of high-priority tasks to preempt low-priority tasks with respect to the cache. Thus, the resulting schedulability test is the same: the total utilization of each cache processor is at most one. We do not need to explicitly evaluate the schedulability of the physical processors, since the set of tasks on each physical processor is a subset of the tasks on the respective cache processor. Thus, if each cache processor is schedulable, then each physical processor is as well.

Ex. 5 (cont'd). Because tasks on processors P_1 and P_2 share the same set of colors (red and green), they form a single cache processor. Since the utilization of the four tasks is one, both the cache and the processors are schedulable. If all tasks required both red and green, then the tasks would necessarily be serialized as if they were on a single processor (assuming cache isolation is required). However, because tasks T_2 and T_4 do not share the same colors, they can execute concurrently, as depicted in Fig. 6.

Note that this schedulability condition is only sufficient and may be pessimistic: it may be possible for tasks to run concurrently on different physical processors if they require disjoint colors. This pessimism can be avoided in two ways: through tighter analysis of the aforementioned more general scheduling problem, or by assigning colors to tasks in a way that reflects our uniprocessor analysis, as discussed later.

3.2 Related Work

Prior work exists on cache management that is similar to ours in some respects. An approach called *cache lockdown* (not to be confused with our use of the 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 (see [14] for an overview). 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 application source code. However, it may be difficult to size partitions so that the cache is efficiently uti-

lized from a system-wide perspective.

In general, the problems of optimally assigning tasks to processors and colors to tasks are both NP-hard in the strong sense [5], though suboptimal heuristic-based algorithms have been proposed and evaluated on different hardware platforms [9, 22, 24]. Cache partitioning is actually a special case of our cache locking approach in which inter-task cache conflicts are entirely eliminated, and hence the usage of a locking protocol is obviated. However, our generalization allows more flexibility in determining color assignments and can enable greater dynamism at runtime.

A system execution model called PREM [25] has been proposed that takes an approach similar to ours. Specifically, PREM uses scheduling to reduce or eliminate contention for shared resource accesses, including main memory. However, unlike our proposed cache management techniques, PREM is restricted to single-core systems. An extension to PREM [31] exists that uses memory-centric TDMA-based scheduling on multicore processors, but assumes no shared resources among cores except memory—each core's last-level cache is private.

In work on timing analysis, methods pertaining to memory hierarchies have been proposed (*e.g.*, see [15, 26] and the references therein). Related hardware-based techniques include *cache bypass* [11], which reduces cache conflicts by exploiting special hardware instructions, and methods for making multicore platforms more predictable at the processor [23] and interconnect levels [1, 10, 27, 28]. In contrast to this work, our approaches are software-based.

4 Implementation

We implemented the cache management techniques described above in an MC^2 prototype within LITMUS$^{\mathrm{RT}}$ [18]. In developing this implementation, we restricted attention to tasks that are independent (no shared resources other than cache lines), have only memory-resident pages, and that do not share pages with each other. We used the ARM machine described in Secs. 1-2 as our development platform. All page coloring was done with respect to the last level of cache, which on this machine is its L2. In prior work, we found that level-B schedulability is greatly improved if one CPU is designated as a *release master* that processes all job-release interrupts and is not assigned any level-B tasks [12]. We employed a release master in our current prototype, so level-B tasks are actually only scheduled on three CPUs (level-C tasks can execute on the release master).

As noted earlier, we implemented coloring with respect to tasks' data pages. We leave the coloring of other areas of memory as future work. We believe this is reasonable for a first prototype, as our implemented tasks have a small per-job code footprint that does not make any system calls and operate only on the colored memory they allocate. We implemented a memory allocation function similar to `malloc` that modifies the page tables of the backing user process for each task to map pages with the proper colors into its contiguous virtual address space.

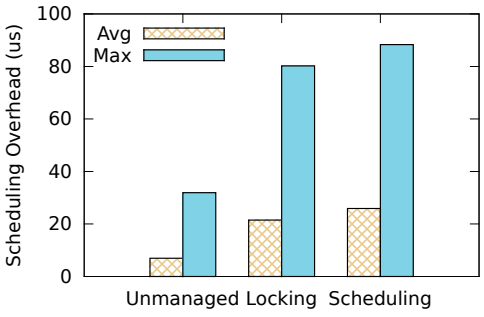
Controlling eviction. When discussing various page-coloring-oriented cache management techniques in Sec. 2, we assumed that the OS can precisely control the cache sets and ways that a task's data is loaded into. In our implementation, this functionality is achieved by exploiting cache lockdown using a clever method proposed by Mancuso, *et al.* [19]. As noted earlier, *cache lockdown* allows certain ways to be marked as unavailable for allocation (or locked down) such that the contents of a locked way cannot be evicted.

The approach in [19] utilizes a variant of cache lockdown called *Lockdown by Master* (*LbM*), where each CPU P_q has access to a per-CPU lockdown register[8] x_q such that bit i in x_q is zero if allocation can occur in way i for memory references from CPU P_q, and one otherwise. LbM is supported on our ARM platform. As noted by the authors of [19], setting all but one bit of x_q to one pigeonholes memory requests from CPU P_q to be allocated in a specific cache way. By applying this idea, memory can be *prefetched* by reading it in a "prefetch" loop such that the loop code occupies at most one cache line and is cache-line aligned. During prefetching, CPU-local interrupts must be disabled to avoid interrupt-related cache pollution. To ensure that each prefetched cache line is read into the proper way, none of the memory to be prefetched can be cached elsewhere. Consequently, under cache locking when a job releases its color locks, and under cache scheduling when a job completes or is preempted, all prefetched pages are *flushed* or evicted from the cache. This is done using a similar process to prefetching, except that pages reserved for flushing are loaded into the ways to be flushed thereby evicting the job's cached pages.

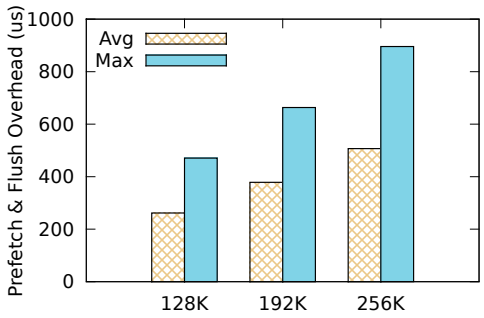
Evaluation. We evaluated our implementation by measuring system overheads (including cache prefetching/flushing costs) and observed WCET improvements. To obtain these measurements, we traced the behavior of task sets with varying task counts, where periods and utilizations were generated uniformly from $\{25, 50, 100, 200\}$ms and $[0.01, 0.05]$, respectively. Level-B tasks were assigned to cores using the worst-fit heuristic. (Any task set that could not be so assigned was discarded.) Cache prefetching was turned off. Each task was defined via a per-job code sequence in which it reads its WS in a random order one or more times.

Scaling factors. A job executed under cache locking or cache scheduling does not suffer cache misses and therefore completes execution earlier than it would in a system without cache management. We use *scaling factors* to denote the ratio of per-job execution times without and with cache management, *e.g.*, if cache scheduling reduces a task's per-job execution time from 10 ms to 2 ms, then this task has a scaling factor of five. The worst-case scaling factors on our test platform were on the order of 3.5 to 4.5, while average-case scaling factors were between 3.3 and 3.9.

[8] Actually, there is a per-CPU register for instructions and data, but we are concerned with only data at this point.



(a) Scheduling overhead.



(b) Prefetch & flush overhead.

Figure 7: Each graph shows average-case (orange, textured) and worst-case (blue, solid) measurements. **(a)** Scheduling overheads. **(b)** Prefetch and flush overheads for various WSSs.

In Sec. 5, we use these scaling factors to compare schedulability with and without cache management in the presence of system overheads. While it might be preferable to use WCETs predicted by timing analysis tools in such comparisons, adequate tools for multicore platforms do not yet exist. Also, note that observed WCETs lower bound predicted ones (if the prediction is safe). Thus, observed values give some indication of how a tool might perform given varying degrees of information about cross-core cache interactions.

System overheads. Cache management increases scheduling costs and consequently the duration of OS overheads interrupting task execution. Additional overhead arises under cache locking due to locking-protocol overhead, and under cache scheduling due to added complexity for maintaining cache processor state. Fig. 7(a) depicts scheduling overheads (*i.e.*, the time taken by the OS to make a scheduling decision) for level B assuming no cache management (*Unmanaged*), cache locking (*Locking*), and cache scheduling (*Scheduling*). Observe that worst-case overheads were increased by around 60 μs for cache scheduling and 50 μs for cache locking.

The cache lockdown prefetching and flushing operations must also be considered. A task will not begin execution until its assigned processor has cached the entirety of its WS. Additionally, whenever a job unlocks or is preempted from the cache, its WS must be flushed from the cache. In Fig. 7(b), we show the cost of prefetching and flushing for different WSSs. The worst-case cost varied from 400 μs to 900 μs in our experiments, depending on WSS. With respect to schedulability, these costs and higher scheduling overheads are offset by lower WCETs as we show next.

5 Schedulability Study

In this section, we evaluate the utility of our proposed cache management techniques from a schedulability perspective, with measured overheads considered, and examine the tradeoff between improved WCETs enabled by our techniques and any utilization loss due to cache management.

Utilization scaling. To evaluate the schedulability of a task system using cache management, we scale the level-B utilization of tasks scheduled without cache management by a scaling factor commensurate with those observed in Sec. 4. This utilization scaling theoretically allows a scheduler using our cache management techniques to schedule task systems that, unmanaged, contain tasks with level-B utilizations greater than one or have a total level-B utilization greater than the number of processors. For example, using our cache management techniques, it may be possible to schedule on four processors a task system with a total unmanaged level-B utilization of five and that contains a task with an unmanaged utilization of 1.5. These observations motivate the experimental design of of our schedulability study, as is discussed next.

We evaluated the schedulability of randomly generated task systems having equal level-B and -C subsystem utilizations. That is, for each system utilization x, we generated task systems where the sum of all level-B tasks' level-B utilizations was equal to x, and the sum of all level-B and -C tasks' level-C utilizations together also summed to x (recall that in mixed-criticality scheduling, a task has an execution cost, and hence utilization, for each criticality level). The level-C utilization of each level-B task was assumed to be 10% of its level-B utilization. We considered task systems with utilizations without cache management of $\{1.0, 1.1, \ldots, 10.0\}$ with respect to the previously described system (four processors, 1MB 8-way set associative cache, with 512 KB allocated to each criticality level). Both the level-B utilizations of level-B tasks and the level-C utilizations of level-C tasks were uniformly distributed over [0.1,0.4] or [0.5,0.9], though in all graphs presented herein, which depict relevant trends, the former utilization distribution is assumed.[9] We note that greater per-task utilizations could be supported using cache management (*e.g.*, 1.5). However, we did not consider such systems in our evaluations as there is no basis for comparison with a system with an unmanaged cache (which cannot schedule tasks with utilizations greater than 1.0).

Schedulability was determined by evaluating the generated task systems using the level-B and -C schedulability conditions given in [20], with overheads factored in using the techniques described in [4, Chap. 3]. Worst-case

[9]The remaining graphs are available in an online appendix available at `http://www.cs.unc.edu/~anderson/papers.html`.

(average-case) overheads were assumed for level B (C), as level B (C) is HRT (SRT). Comparisons between systems with and without cache management were performed by generating task systems for an unmanaged system, and then scaling task execution times by scaling factors commensurate with those observed in Sec. 4.

Coloring schemes. We investigated two heuristics for assigning colors to level-B tasks. Each was designed to reduce cross-core color contention, since such contention is detrimental to both cache locking and cache scheduling. Both heuristics function similarly and are applied assuming that tasks have been previously assigned to physical processors and that physical memory pages are unconstrained (conceptually, there are infinitely many physical memory pages available for each color—note that our test platform has 2^{13} pages per color). Letting S denote the size of the cache and W denote the maximum WSS, we divide the cache into $\lfloor S/W \rfloor$ "bins", each of size W. Under *way-first* (*color-first*) *allocation*, the number of ways (colors) in each bin is maximized. For example, a bin half the size of an 8-way, 32-color cache has 8 ways of 16 colors under way-first allocation, and 4 ways of 16 colors under color-first allocation. Under either heuristic, processors are packed into the obtained bins using the worst-fit heuristic, and each task is assigned colors to satisfy its WSS from the set of colors assigned to the processor on which it is partitioned. Note that if $W \leq S/m$, then our heuristics partition the cache.

These heuristic are more flexible than cache partitioning, as they allow colors to be shared across processors. However, the problem of assigning colors to tasks is generally intractable and may be particularly hard if the number of physical memory pages per color is severely constrained. In many embedded systems, such constraints may exist due to SWaP requirements, the need to support separate processing modes,[10] *etc.* Nonetheless, the experiments below demonstrate that if a reasonable color assignment can be found, then schedulability-related impacts can be dramatic. We plan to further investigate color assignment strategies in the future to widen the applicability of our work.

Schedulability. We now discuss several schedulability graphs involving randomly generated task systems. A cache management scheme's *schedulability* is specified with respect to a set S of generated task systems and is defined as the fraction of S deemed schedulable under that scheme. We present level-B (HRT) and level-C (SRT) schedulability graphs in Fig. 8. In the level-B graphs, schedulability is plotted versus total level-B utilization *prior to scaling*. In the level-C graphs, it is plotted versus total level-B and -C utilization assuming level-C execution costs for both. With respect to the level-B scheduling policy, we denote ordinary partitioned RM (P-RM) scheduling with an unmanaged cache (*i.e.*, no cache locking or scheduling) as *UM*, cache locking as *CL*, cache scheduling as *CS*, and cache locking with period splitting as *CL-PS*. We do not present graphs for job splitting as it was always inferior to period splitting alone. To prevent interference between tasks of different criticality levels, level-B and -C tasks are each assigned half of the available cache, so a WSS of 256K corresponds to half of the available level-B cache. Our major observations are as follows.

Obs. 1. In all observed cases, at least one of the proposed cache management solutions (CL, CS, or CL-PS) offered improved level-B schedulability, often scheduling task systems with almost 50% greater processor utilization than an unmanaged P-RM level-B subsystem (UM).

This observation is corroborated by insets (a)-(e) of Fig. 8, which depict level-B schedulability. For example, in Fig. 8(a), with a scaling factor of three, which is less than we observed in practice, cache scheduling (CS) could schedule task systems with a level-B utilization of 50% more than that of an unmanaged P-RM system (UM). By scaling level-B tasks' level-B execution times down (to account for fewer cache misses due to cache management), we are able to schedule task systems that previously would have been unschedulable because they would have overutilized the available processing cores. This effect is even more pronounced with larger scaling factors. These results show that the benefits of cache management techniques may outweigh the increased overheads they entail.

Obs. 2. For some system configurations, unmanaged P-RM (UM) offered improved level-B schedulability over one or more cache management techniques. This suggests that the cache can be managed *improperly*.

In the level-B schedulability results presented in insets (a)-(d) of Fig. 8, unmanaged P-RM (UM) outperforms cache locking (CL) in many cases and in Fig. 8(d), unmanaged P-RM (UM) outperforms cache scheduling (CS). However, as described in Obs. 1, at least one of our cache management solutions outperforms unmanaged P-RM (UM). Thus, a system designer must evaluate their task system to decide which of our cache management solutions is best.

Obs. 3. In all observed cases, period splitting improved level-B schedulability under cache locking.

This can be seen in insets (a)-(e) of Fig. 8. Henceforth, when we refer to cache locking, we assume period splitting (CL-PS) is used.

Obs. 4. Under color-first allocation, cache locking (CL-PS) offered the best level-B schedulability, while under way-first allocation, cache scheduling offered the best level-B schedulability. This suggests that color assignment has significant impact on level-B schedulability.

Comparing level-B schedulability using way-first and color-first allocation in insets (b) and (d) of Fig. 8, respectively, we observe that cache locking (CL-PS) performs much better under color-first allocation. This is because the blocking bound for cache locking is $O(mr/k)$ where r is the maximum number of ways of a color required, which is minimized under color-first allocation, and k is the to-

[10] When assigning colors to the tasks that participate in one mode, the execution requirements of tasks that participate only in other modes can be ignored, but those tasks still consume memory pages and hence colors.

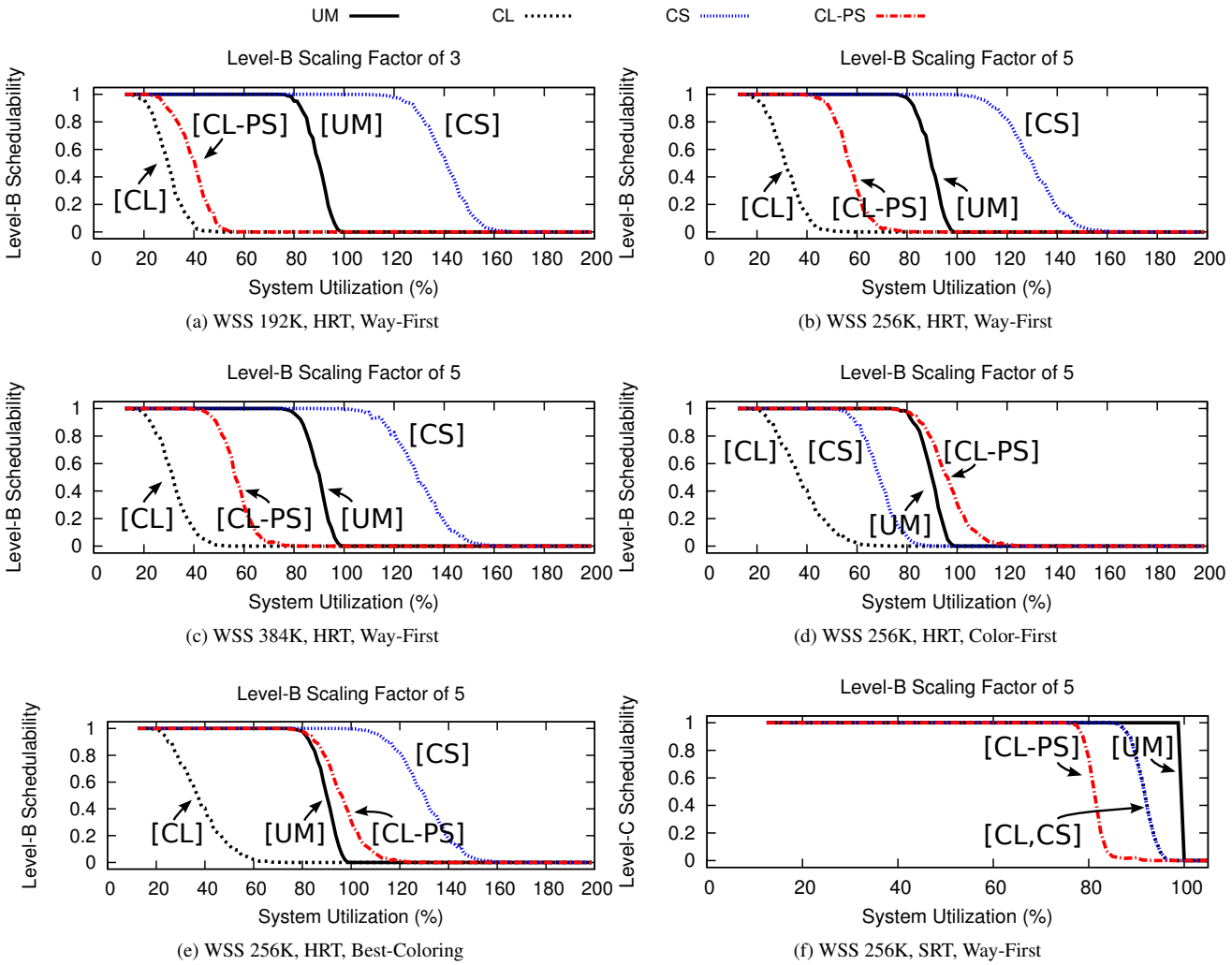


Figure 8: Insets **(a)**-**(d)** compare the proposed cache management approaches in terms of level-B schedulability for different scaling factors, WSSs, and coloring algorithms. Inset **(e)** compares the schedulability of the proposed cache management techniques using the coloring algorithm that provides the best schedulability for each technique, respectively. Inset **(f)** depicts level-C schedulability.

tal number of ways. In contrast, cache scheduling has much better level-B schedulability under way-first allocation, but performs comparatively worse under color-first allocation. This is because of the pessimism of our analysis, which assumes that only one task executes with a color at a time, regardless of the number of ways required (recall from Sec. 3 that tighter analysis for cache scheduling is an open problem). This pessimism is particularly detrimental to level-B schedulability using color-first allocation. However, under way-first allocation, tasks use all of the ways of comparatively fewer colors, which in turn reduces the number of cache processors, and improves schedulability. These observations demonstrate the impact that the coloring algorithms have on level-B schedulability.

Obs. 5. The overheads associated with our cache-management techniques only minimally impacted level C.

Our techniques have a profound impact at level B. However, as can be seen in Fig. 8(f), schedulability at level C is decreased by a comparatively small amount. While this offset means we can schedule no additional level-C work, we could execute many more high-criticality level-B tasks in lieu of lower-criticality level-C tasks.

Obs. 6. In all observed cases, cache scheduling under way-first allocation provided better level-B schedulability than any other cache management technique under either way-first or color-first allocation.

This can be seen in insets (a)-(e) of Fig. 8. Under color-first allocation, as in Fig. 8(d), level-B schedulability under cache scheduling (CS) was less than that of cache locking (CL-PS). However, in all cases in which cache locking (CL-PS) outperforms cache scheduling using color-first allocation, level-B schedulability using cache scheduling under way-first allocation was greater. This can be seen in Fig. 8(e), which shows for each technique the best level-B schedulability seen under either coloring scheme.

Cache scheduling also has practical benefits over cache locking. Cache scheduling is easier to implement, being little more than a multiprocessor RM algorithm with an extra

"mapping" step, while cache locking uses both partitioned RM and a locking protocol. Additionally, cache scheduling is more flexible than cache locking (which uses period splitting). For example, it can be more easily applied to schedule non-periodic task systems. Thus, unless a system is unable to use way-first page allocation, cache scheduling is the recommended cache-management technique.

Future work. This work opens many avenues for future research, which we plan to pursue, ranging from systems-level issues to theoretical scheduling problems. For example, we plan to explore cache management techniques for shared libraries, dynamic memory, and jobs with WSSs larger than the cache. We also plan to evaluate, from the perspective of rigorous timing analysis, the improvements to calculated WCETs made possible by these cache management techniques. Additionally, we are interested in developing scheduling algorithms, schedulability tests, and resource allocation techniques that can support both greater system utilizations (with respect to both the processors as well as the cache) and more than two criticality levels.

6 Conclusion

We have presented several techniques for managing shared caches on multicore systems within the MC^2 mixed-criticality scheduling framework. We also presented experimental results obtained from an implementation of these techniques on a quad-core ARM machine. These experiments indicate that proper shared cache management can lessen WCETs and positively impact schedulability despite increased system overheads. In fact, in our experiments, task systems could be scheduled successfully with cache management that had total utilizations exceeding the capacity of our test platform by 50% or greater without cache management. The development of these cache management techniques as well as MC^2 was motivated by ongoing work with colleagues at Northrop Grumman Corp. (NGC) on defining useful real-time resource allocation infrastructure for future UAVs. These UAVs represent an interesting challenge problem in work on cyber-physical systems.

Acknowledgment: We thank Prakash Sarathy, John Scoredos, and Dan Johnson of NGC for helpful discussions concerning the needs of future UAVs and the importance of cache management.

References

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

[2] S. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *ECRTS '12*.

[3] S. Baruah, H. Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS '10*.

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

[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] U. Devi. *Soft Real-Time Scheduling on Multiprocessors*. PhD thesis, University of North Carolina, Chapel Hill, NC, 2006.

[8] P. Ekberg and W. Yi. Bounding and shaping the demand of mixed-criticality sporadic tasks. In *ECRTS '12*.

[9] M. Fernàndez, R. Gioiosa, E. Quiñones, L. Fossati, and F. Cazorla. Assessing the suitability of the NGMP multi-core processor in the space domain. In *EMSOFT '12*.

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

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

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

[13] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *RTSS '09*.

[14] D. Kirk. SMART (strategic memory allocation for real-time) cache design. In *RTSS '89*.

[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*.

[16] J.Y.-T. Leung and C.-L. Li. Scheduling with processing set restrictions: A survey. *International Journal of Production Economics*, 116:251–262, Dec. 2008.

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

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

[19] R. Mancuso, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time colored lockdown for cache-based multi-core architectures. In *RTAS '13*.

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

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

[22] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *EDCC '12*.

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

[24] M. Paolieri, E. Quiñones, F. Cazorla, R. Davis, and M. Valero. IA^3: An interference aware allocation algorithm for multicore hard real-time systems. In *RTAS '11*.

[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*.

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

[27] 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*.

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

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

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

[31] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48:681–715, 2012.