Real-Time Cache Management Framework for Multi-core Architectures

Renato Mancuso[†], Roman Dudko[†], Emiliano Betti[*], Marco Cesati[*], Marco Caccamo[†], Rodolfo Pellizzoni[‡]
[†]University of Illinois at Urbana-Champaign, USA, {rmancus2, dudko1, mcaccamo}@illinois.edu
[*]University of Rome "Tor Vergata", Italy, {betti, cesati}@sprg.uniroma2.it
[‡]University of Waterloo, Canada, rpellizz@uwaterloo.ca

***Abstract*—Multi-core architectures are shaking the fundamental assumption that in real-time systems the WCET, used to analyze the schedulability of the complete system, is calculated on individual tasks. This is not even true in an approximate sense in a modern multi-core chip, due to interference caused by hardware resource sharing. In this work we propose (1) a complete framework to analyze and profile task memory access patterns and (2) a novel kernel-level cache management technique to enforce an efficient and deterministic cache allocation of the most frequently accessed memory areas. In this way, we provide a powerful tool to address one of the main sources of interference in a system where the last level of cache is shared among two or more CPUs. The technique has been implemented on commercial hardware and our evaluations show that it can be used to significantly improve the predictability of a given set of critical tasks.**

I. Introduction

Modern systems are at a turning point as multi-core chips become mainstream. Multi-core architectures are shaking the very foundation of modern real-time computing theory, i.e. the assumption that worst case execution time (WCET) can be calculated on individual tasks to compute the schedulability of the complete system when tasks are running together. This fundamental assumption has been broadly accepted by classic scheduling theory for the past three decades; unfortunately, it is not even true in an approximate sense in a modern multi-core chip, and this leads to a lack of composability. Shared hardware resources like caches, main memory, and I/O buses are all sources of unpredictable timing behaviors and temporal dependencies among contending real-time tasks. In this work, we focus on cache-based multi-core architectures and we provide a solution to the problem of efficiently exploiting the benefits of cache memory without paying the penalty of non-deterministic temporal behavior. In fact, CPU caches sharply improve average case task performance, but their behavior leads to several issues when calculating WCET bounds for hard real-time tasks [1, 2].

In commercial multi-core systems, contention for allocation of memory blocks in last level cache is a major source of unpredictability. We distinguish four main types of cache interference: a) intra-task interference, occurring when a task evicts its own cached blocks; b) inter-task interference, occurring whenever a task evicts cached blocks of another task scheduled on the same CPU; c) cache pollution caused by asynchronous Kernel Mode activities, such as Interrupt Service Routines (ISR) and deferrable functions; and d) inter-core interference, occurring when two tasks running on different cores evict each other on a shared cache. While the first three types of interference occur in both single-core and multi-core systems, the last one creates inter-core dependency which makes traditional schedulability analysis not applicable. Due to all the different sources of interference that exist in a multi-core system, enforcing a deterministic behavior on traditional caches means making them operate in a restrictive manner. Various solutions (see Section II) have been proposed in the literature to address these different scenarios; however, to the best of our knowledge, there is no available software mechanism that can simultaneously address all four sources of interference while efficiently utilizing the available cache resources. For example, core-based cache partitioning suffers from inter/intra-task interference; blindly locking memory areas in cache is highly inefficient because in the worst case no more than W lines can be locked in a W-way set associative cache.

We propose an integrated solution. Our solution is divided in two stages that address two different but highly coupled problems. First, we propose to profile critical real-time tasks to determine what their memory access patterns are. We perform the memory analysis by running each task in a test environment, recording all the memory accesses and generating a "profile file". This profile is execution-independent, i.e. it is independent from the particular set of virtual memory addresses assigned by the kernel at execution time. Such profile file, generated offline, can be used to drive the second stage (deterministic cache allocation). The execution-independence allows it to be used to drive the second stage an arbitrary number of times on the same or on a *compatible* system (see Section IV).

Secondly, we propose a novel, highly efficient deterministic cache allocation strategy called "Colored Lockdown". The key idea of the proposed *Colored Lockdown* is to combine two techniques (coloring and lockdown, see Section II) that have been proposed in the past, exploiting the advantages of both of them but mitigating their disadvantages. In particular, we use (a) coloring to optimize the packing in cache of those memory pages which are frequently accessed (this can be done by re-arranging physical addresses); then, we use (b) lockdown to override the behavior of the cache replacement policy to make sure that what has been allocated, on behalf of a given task, will not be evicted while that task is running. It is important to note that, in the proposed solution, cache allocation is deterministically controlled at the granularity of a single memory page and that it is independent from the specific cache replacement policy.

We have successfully implemented and tested the proposed framework on a commodity ARM-based development board and we have evaluated the porting effort on a Freescale P4080 platform. It is worth to note that the proposed cache management framework can be exploited requiring no hardware modification. Moreover, similarly to what we did on the P4080

978-1-4799-0187-6/13/$31.00 ©2013 IEEE

platform, the proposed technique can be applied to a variety of platforms currently available on the market. Finally, unlike many works that have been presented in the past, our technique works on both data and code regions [3, 4, 5]. Experimental results show that, in the considered benchmarks, eliminating the interference in the last level cache can lead up to a 250% improvement of the execution time, with a resulting cache occupation as small as 4 memory pages.

The main contributions of this paper are:

- Providing a complete framework to analyze the memory access patterns of a given task, generating a memory profile which can be used to perform deterministic cache allocation. This can be done with minimal effort and without impacting schedulability or certifiability.
- Developing a novel kernel-level, efficient and deterministic cache allocation technique which can be used in commodity embedded multi-core systems. Such technique can be operated regardless of the cache replacement policy in use and without any hardware modification, to meet hard real-time constraints.
- Showing experimentally that the problem of finding an optimal cache allocation, when the cache size is limited, can be treated as a convex optimization problem. In fact, the performance curves, obtained when incrementally locking memory pages according to the generated profile, can be fit with convex functions to exploit well known convex optimization algorithms.
- Providing a cache management framework that improves execution predictability of critical real-time tasks in a multi-core environment by enforcing a 100% cache hit rate on the most "hot" (frequently accessed) memory pages. Such a framework can also be exploited by static analyzers to compute tighter WCET bounds.

The rest of the paper is organized as follows. First, Section II recalls some background principles and reviews prior research related to profiling and cache management techniques. Section III introduces the considered task model. Next, Section IV illustrates the proposed memory profiling technique. Section V explains how Colored Lockdown works. The presented results in Section VI show how the proposed framework can improve the schedulability of a set of critical tasks by eliminating last level cache interference. Finally, the paper concludes in Section VII.

II. Background and Related Work

In this section we will briefly discuss existing profiling and cache management techniques and we will explain how our cache allocation strategy differs from existing related work.

A. Task execution and memory profiling

One method that can be deployed to understand the behavior of a task, as well as its memory requirements, is profiling. The key idea is to observe the actions that the task itself performs while executing. The profiling approach is an alternative technique over static analysis when dealing with complex architectures that are difficult to model due to inherently non-deterministic components. Profiling can be performed by adding instrumentation instructions to the original code of the task. Instrumentation tools are usually classified in two categories: source-level and binary. Binary instrumentation adds extra instructions to a compiled program, before it is executed (e.g. QPT [6], Pin [7]) or while the program is running (e.g. Valgrind [8]). In the source-level approach, instrumentation is performed at compile time. Existing automatic source-level instrumentation is done either by performing source-to-source translation (e.g. ROSE [9]), or by introducing additional specific compilation logic [10, 11].

Since our target is to understand the memory access pattern of a given task, the chosen solution involved using a tool which adds instrumentation at run-time. Specifically, as a part of our proposed framework, we have used Valgrind (the Lackey tool) to perform the first step of the analysis. The advantage of this solution is twofold. First, it does not cause a difference in the memory access patterns of the task under analysis when switching from the profiling environment to the actual execution. In this way, we are able to trace any memory access performed by the task in a non-intrusive manner. Second, since Valgrind is available for a large number of architectures, it does not limit the portability of our technique.

To the best of our knowledge, no solutions have been proposed before to create a memory-page-level detailed profile of an application which is, as previously stated, execution-independent and that can be used to optimize system cache allocation at run-time.

B. Cache management

Modern commercial CPUs feature at least one cache level organized as a write-back, W-way set associative cache. Depending on the addressing patterns of processes running on the system, data is loaded into and written-back from cache in units known as *cache lines* (or *cache blocks*). A W-way set associative cache is divided into W *cache ways*. Each data block can be loaded in any way, chosen at fetch time by the cache controller according to the replacement policy [12].

Once the cache way has been selected, the exact position inside the way is given by the value of a subset of the bits which compose the data address, called the *index*. An *associative set*, or simply *set*, is the set of all the cache lines (one for each of the W cache ways) with the same index. *Tag* bits are used to detect hits, while *offset* bits are used to address a particular byte in a cache block. The resulting address structure is shown in Figure 3a in Section V.

If the cache controller considers as tag (index) the value of some bits in the *physical* address of the fetched data, then we say that the cache is *physically* tagged (indexed). Otherwise it is called a *virtually* tagged (indexed) cache [12]. In multi-core systems, shared caches cannot work with virtual addresses, thus, modern COTS systems are increasingly being built featuring physically indexed, physically tagged caches.

In order to mitigate the inherently non-deterministic cache behavior, several different techniques have been proposed to directly or indirectly control cache allocation. They can be grouped in three main categories: cache partitioning, cache locking, and page coloring [13].

1) Cache partitioning: The idea behind cache partitioning is to assign a given portion of cache to a given task or core in the system to reduce cache pollution. Similar techniques can be applied in software [3, 14, 15, 16, 17] or in hardware [18, 19, 20] with different granularities. Software partitioning techniques usually rely on an indirect control over the cache, manipulating address-to-line mapping at OS, compiler, or application level. However, they are not easily applicable in a system-wide fashion. On the other hand, hardware-based techniques require additional fine-grained platform support and, since they are typically core-based, they are not suitable to resolve inter/intra-task interference.

2) Cache lockdown: Commercial cache controllers usually implement a simpler content management paradigm: cache lockdown. Locking a portion of the cache means excluding the contained lines from the cache replacement policy, so that they never get evicted over an arbitrary time window. Lockdown is a hardware-specific feature, which typically is done at a granularity of a single line or way.

Because the number of ways W is usually limited (in the range from 4 to 32), mechanisms that can lock down a whole cache way have not been explored deeply. Conversely, techniques which exploit "lockdown by line" mechanisms have been extensively studied [21, 22, 23, 24]. However, the "lockdown by line" strategies provided by most of the current commercial embedded platforms are non-atomic. This makes it difficult to predict what is cached and what is not. Moreover, multi-core shared caches are usually physically indexed (and tagged). Thus, if no manipulation is enforced on the physical addresses of the locked entries, in the worst case no more than W locked lines could be kept at the same time.

3) Page coloring: A particular case of software-based cache partitioning is page coloring [25, 26, 27, 28, 29]. In this case, partitioning is enforced at the granularity of a memory page, and can be done by manipulating the virtual-to-physical page translation at the OS level. Some coloring strategies have been shown to improve average case performances [28, 29, 26, 30].

However, this approach shares a fundamental problem with cache partitioning: if a non-deterministic replacement policy is implemented, then self-evictions could always occur, hence in the worst case one single cache way is exploited. In addition, coloring kernel pages and data accessed by Interrupt Service Routines (ISRs) can be difficult, especially due to hardware-specific constraints which force some structures to be placed at hard-coded offsets in physical memory. This leads to a lack of predictability and, again, to pessimistic WCET bounds.

In our proposed Colored Lockdown, we use coloring in a completely different way: we do not allocate a particular color to a given entity of the system, but we instead color pages to make sure that a given page will map to a specific cache set. This allows us to efficiently pack all the most frequently accessed memory pages of each hard real-time process in cache and to perform selective lockdown. As a result, system designers can be confident about which memory area accesses of a real-time task will always trigger a cache hit, no matter what other tasks are running on the system. Thereby, using the proposed hard real-time cache management framework results in less pessimistic WCET bounds for critical tasks.

III. Task model

We consider a system with mixed criticalities, where both critical and non-critical tasks can be active at the same time. For the sake of simplicity, we assume a partitioned, priority-based scheduler, even if this work can be easily extended to systems having a global scheduler. In our model, non critical tasks are considered as black boxes, and we do not enforce any constraint on them. On the other hand, we assume that critical tasks are periodic and have an initial start-up phase executed once for initialization, and a series of periodically released jobs.

Moreover, for the profiling procedure to produce a valid output, an additional constraint is required: that the task under analysis does not perform any dynamic memory allocation after the start-up phase. This is because dynamic allocations can trigger the creation of a new memory region. As we will show, we need the set of memory regions detected at the end of the start-up phase to remain stable throughout the task execution, and we need that the number and the order of such memory regions is unaltered from execution to execution. Note however that when dealing with real-time systems, this assumption is generally satisfied, since dynamic memory allocations are a source of unpredictability and are strongly discouraged for certification purposes [31].

IV. Memory profiling

The aim of the memory profiler is to identify the "hot" (most frequently accessed) virtual memory pages for a given executable. However, the detection procedure can not be based on absolute virtual addresses, because virtual addresses change from execution to execution. A common abstraction in operating systems is the concept of a *memory region*: a range of virtual addresses assigned to a process to contain a given portion of its memory. Thus, processes own several memory regions to hold executable code, the stack, heap memory and so on. Our goal is thereby exploiting such an abstraction to create a profile where a hot page is expressed as an offset from the beginning of the memory region it belongs to. In this way, the profile needs to be created only once, and it is possible to determine the effective address of the hot page at execution time. For instance, the hottest memory page might be the fifth page (offset) inside the third memory region of the process. In this case, the corresponding entry in the memory profile will look like: `3 + 0x0005`.

Memory pages belonging to dynamically linked libraries are not included in the produced profile. Thus, if it is necessary to keep track of the memory accesses performed by a given task inside a library procedure, that library has to be statically linked inside the task executable.

Note that in the presented implementation we have generated the profiles considering a single input vector for each task. However, as a future work, we plan to improve the profiling technique to aggregate the data acquired from several task executions with different input vectors to enhance coverage.

If the assumptions made in the previous section hold, then, from the memory profile, it will always be possible to correctly calculate the position of each hot memory page. Finally, the obtained profile can then be handed off to the Colored Lockdown module, which can process this information and perform the desired operations on the correct set of pages.

A. Approach description

The proposed profiling technique can be divided in three phases. In the first phase, called "accesses collection", the following operations are performed:

1) collect all of the virtual addresses accessed during the task execution;
2) create a list of all the accessed memory pages sorted by the number of times they were accessed;
3) record the list of memory regions assigned by the kernel in the profiling environment.

In the second phase, called "areas detection", the analyzed task is run outside of the profiling environment, so that we can:

4) record the memory areas assigned by the kernel under normal conditions.

Finally, in the third phase, called "profile generation", we:

5) link together the list of memory regions assigned during the profiling (obtained in step 3) with those owned by the process under normal conditions (step 4);

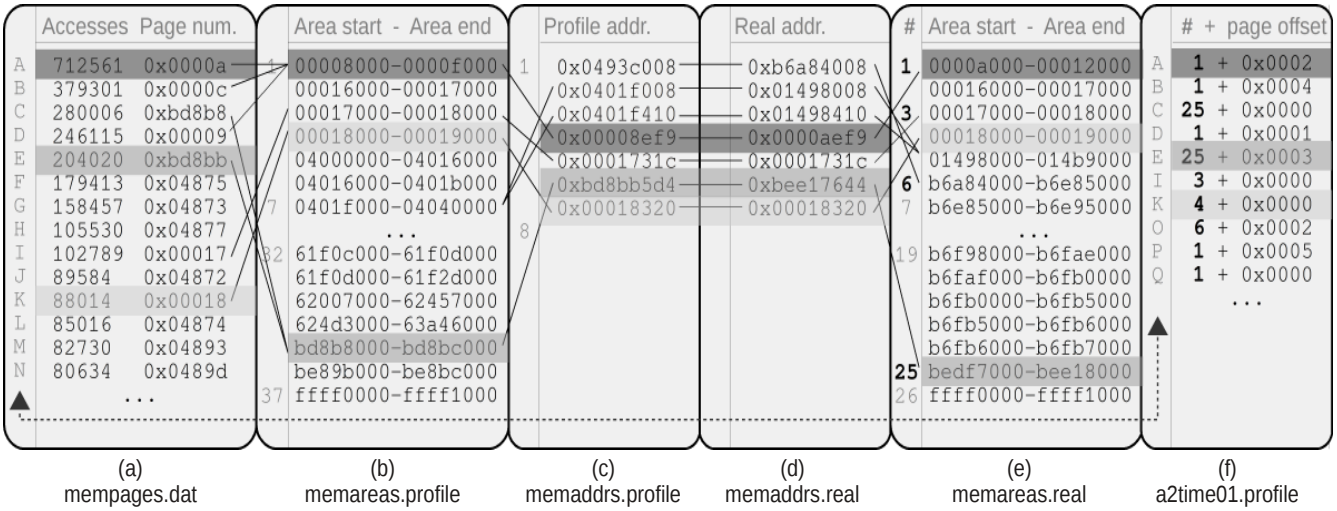

Fig. 1. Profile creation for the benchmark "a2time01" and relations between the files involved, starting from the ranked list of accessed memory pages (a) to the final profile file (f).

6) determine in which region of the list obtained in step 4 each entry of the list obtained in step 2 falls, according to the mapping determined in step 5;
7) generate the *execution-independent* memory profile of the task.

The profile file obtained at the end of the procedure can be given as an input to drive the Colored Lockdown procedure.

In order to exploit the presented technique on a given executable, we need to insert, in the executable code, a call to a small procedure which generates the intermediate auxiliary files. In particular, it has to be inserted between the end of the start-up phase and the beginning of the periodic phase. In the final version of the executable, or in general once the profile has been generated, the procedure can be maintained (and called) as it is, to avoid altering the arrangement of the text section. Alternatively, its assembly instructions can be replaced with NOPs. In the rest of the paper we will refer to this procedure as: `aux_files_out`.

1) Accesses collection: To collect information on which memory pages are most frequently used, the profiler must first intercept all memory accesses. To perform this step, we used a sub-tool of Valgrind called Lackey, which outputs (virtual) memory accesses as the program is running. Since the granularity of the Colored Lockdown mechanism is one memory page, in the first elaboration step, we trim the 12 least significant bits of the outputted addresses to obtain the (virtual) page number. These page accesses are recorded together with the number of times they are accessed. Doing so, it is possible to sort pages by the number of accesses to produce a ranking of the most frequently used "hot" memory pages.

The list produced in this step is called `mempages.dat`. The entries in this list are shown in Figure 1a, where an alphabetic ranking is associated with the page hotness. However, it contains a list of hot memory pages identified by absolute memory addresses. As previously stated, they have to be converted to relative values with respect to process memory regions.

Valgrind uses a library preload mechanism to instrument running processes. This means that, when an executable runs under Valgrind, it owns a completely different set of memory regions than when it runs outside the profiling environment. Moreover, the tool Lackey also records the accesses to some memory locations belonging to Valgrind itself. For these two reasons, the pages listed in `mempages.dat` have to be mapped back into memory regions owned by the process outside of the profiling environment.

In this access collection phase, the aforementioned `aux_files_out` procedure generates two additional files, called respectively `memareas.profile` and `memaddrs.profile`. The first file, shown in Figure 1b, contains the list of memory regions assigned to the analyzed task in the profiling environment. Each region is reported as a pair of the form $\langle$start address, end address$\rangle$. This list is obtained relying on the standard Linux *proc* virtual file system. Specifically, `memareas.profile` is generated as a copy of the file `/proc/<pid>/maps`.

The second file `memaddrs.profile`, reported in Figure 1c, is generated by outputting the address of a set of variables located in different regions of the executable. Specifically, the addresses of a stack, a text, a data and a read-only data variable. As explained at the end of this section for sake of simplicity, tracing heap addresses requires additional logic.

2) Areas detection: As previously discussed, we need to map all the memory accesses performed by the process in the profiling environment back to the set of memory regions owned by the same executable running under regular conditions. To do this, we perform a second run of the task without Valgrind, in the native environment.

As in the access collection phase, the discussed `aux_files_out` generates a pair of files which are semantically identical to the previous case, in which the values are referred to the executable in its native environment. These files, reported in Figure 1e and Figure 1d, are given the names `memareas.real` and `memaddrs.real` respectively.

3) Profile generation: In this last phase, the actual profile is generated by merging all the pieces of information contained in the set of files previously created, as we show in Figure 1. For the first step, we leverage on the fact that there is a 1:1 correspondence between the lines of the two files `memaddrs.profile` and `memaddrs.real`, since

they have been generated by the same procedure. We exploit this property to establish a mapping between the set of memory regions belonging to the task in the native and in the profiling environment, i.e. between the entries in the files `memareas.real` and `memareas.profile`.

Once the mapping between the two set of memory regions has been computed, it is possible to relate each accessed page in the profiling environment - i.e. each entry of the file `mempages.dat` - to the correspondent area in the set of memory regions assigned to the process in the native environment. Thus, it is finally possible to come out with a relative notation to address a given hot page that will be valid for every task execution.

Finally, the memory profile (Figure 1f) for the executable is generated. Each entry in this file identifies a hot page by a memory region index and an offset. The entries in this list respect the same alphabetical order as in Figure 1a. The Colored Lockdown module will read this file to determine which pages need to be locked. Note that some pages - such as those belonging to dynamically linked libraries - are not present in the final file because they have been filtered out.

B. Heap memory

As previously mentioned in the accesses collection phase, outputting the address of a single variable in each memory region of interest leverages on the assumption that a text variable, a stack variable, and so on, can be only contained in one memory region. Nevertheless, this is not the case with the heap: the heap could actually be covered by multiple memory regions.[1] Thereby, to find the heap memory regions and to make sure that we catch all the other memory regions that are created using `mmap`, we need to do a little extra work. In particular, we compile the executable *wrapping* all `malloc` and `mmap` calls, which is easily done using a GCC extension (`-Wrap` ld's option), and does not involve re-engineering or rewriting the executable source code. The wrapper is implemented so as to intercept all `malloc` and `mmap` calls which create an anonymous memory region (`MAP_ANONYMOUS` flag). Finally, the starting address of every allocated memory area is appended to the already discussed files `memaddrs.profile` and `memaddrs.real` (Figure 1c,d).

C. Cross-platform compatibility

As previously mentioned, the produced memory profile can be used an arbitrary number of times on the same system where it has been produced. Moreover, it can be produced on a cross-platform and moved in the final system, given that the latter is a *compatible* system. In particular, two platforms are compatible if:

1) they are based on the same architecture;
2) the memory region assignment policy is known to be the same (e.g. they run the same Linux kernel);
3) both use the same compilation tool-chain.

It is worth noticing that the proposed technique relies on task abstractions (such as memory regions) which are common to almost all the OSes. This means that the profiling mechanism can be ported to virtually any OS, provided that it offers as an invariant that the order of the memory regions assigned to a task does not vary from execution to execution. This is true for the great majority of UNIX-based systems.

Moreover, the memory access collection is performed using Valgrind and the dynamic memory allocation detection exploits a GCC extension. Both the tools are available for a wide range of platforms, thus we are confident that our technique applies to a variety of architectures, including x86, ARM and PowerPC.

[1]Under certain implementations of `malloc`, memory allocations over a fixed size are put into a new memory section using `mmap` instead of expanding the heap using the `brk` system call.

V. Colored lockdown

The Colored Lockdown approach consists of two distinct phases. In the first phase, called *start-up color/way assignment*, the system assigns to every hot memory page of every considered task a color and a way number, depending on the cache parameters and the available blocks. During the second phase, called *dynamic lockdown*, the system prefetches and locks in the last level of cache all (or a portion of) the hot memory areas of a given task. In our model this procedure can be deferred until the activation of the first job in the task that will address a certain set of hot memory areas, as shown in Figure 2b.

A. Hardware features

Our work can be applied to systems featuring a last level cache organized as a write-back physically indexed, physically tagged W-way set associative cache. Furthermore, we assume that the way size in bytes is a multiple of the memory page size. This is required to enforce memory coloring at a granularity of one memory page. This assumption is generally valid because modern embedded systems typically have pages of 4 KB and caches in the range from 128 KB to 2048 KB, with a number of ways W in the range of 4 to 32.

We also assume that the system provides a set of registers or dedicated instructions to manage the behavior of the cache controller. In particular, to enforce a deterministic behavior in a SMP system (without stalling any CPU), we have to avoid evictions caused by other cores while fetching on a given CPU. This can be done if: (A) there exists an atomic instruction to fetch and lock a given cache line into the cache, or (B) it is possible to define, for each single CPU in the system, the lockdown status of every cache way. The last mechanism is called *lockdown by master* in multi-core systems, and can be thought as a generalization of the single-core equivalent lockdown by way [32].

Platform			Lockdown by	
Name	Cores	Cache size	Master	Line, atomic
TI OMAP4430	2	1024 KB	Yes	No
TI OMAP4460	2	1024 KB	Yes	No
Freescale P4080	8	2048 KB	No	Yes
Freescale P4040	4	2048 KB	No	Yes
Nvidia Tegra 2	2	1024 KB	Yes	No
Nvidia Tegra 3	4	1024 KB	Yes	No
Xilinx Zynq-7000	2	512 KB	Yes	No
Samsung Exynos 4412	4	1024 KB	Yes	No

TABLE I
LOCKDOWN FEATURES ON MULTI-CORE EMBEDDED SYSTEMS

On a dual-core platform featuring a 2-way set associative cache and a controller with a lockdown by master mechanism, we can set up the hardware so that way 1 is unlocked for CPU 1 and locked for CPU 2, while way 2 is locked for CPU 1 and unlocked for CPU 2. This means that: first, a task running on CPU 1 will deterministically allocate blocks

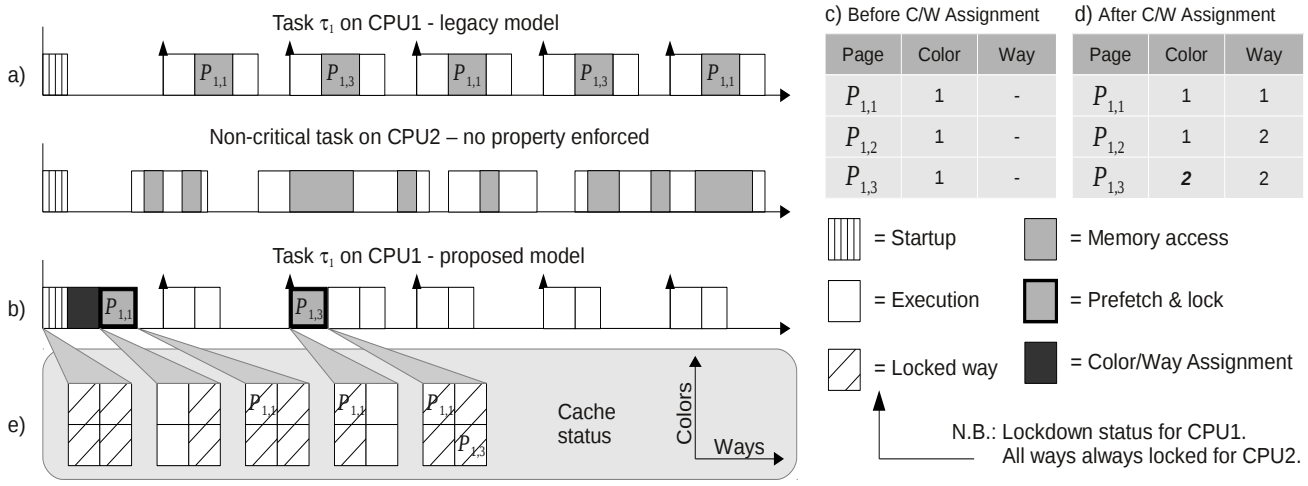

Fig. 2. Comparison between the legacy (a) and the proposed (b) real-time task model on a multi-core architecture.

on way 1; second, blocks allocated on way 1 could never be evicted by a task running on CPU 2. The mirrored situation happens on way 2 referring to CPU 2. This assignment can be easily changed at runtime by manipulating a set of registers provided by the cache controller interface.

If the platform provides an atomic instruction to fetch and lock a cache line, a software procedure that realizes a mechanism functionally equivalent to the lockdown by master can be easily built. Thus, in the rest of this paper we assume without loss of generality that the system provides a lockdown by master mechanism. Table I lists the lockdown features of a few current embedded multi-core systems.

B. Visual example

In the rest of the paper, we will use N to indicate the number of critical real-time tasks in the system with decreasing priority, so that $\Gamma = \{\tau_1, \ldots, \tau_N\}$ is the critical task set. Each task τ_i $(1 \leq i \leq N)$ is composed of a start-up phase s_i and a sequence of jobs $\tau_{i,1}, \tau_{i,2}, \ldots$. Each job $\tau_{i,j}$ is preemptive. Furthermore, we define as M_i the set of the $m_i = |M_i|$ hot memory pages addressed by the jobs in τ_i. As previously discussed, the set of hot memory pages can be determined at the end of the start-up phase, when the Colored Lockdown procedure is invoked, relying on the corresponding profile file.

To show an example, we now refer to Figure 2 to show how this mechanism can work on a dual-core system featuring a 2-way set associative shared cache whose total size is $S_c =$ 16 KB (4 pages). The system is using a partitioned scheduler to bind τ_1 to CPU 1 and a non-critical task to CPU 2. No assumptions are made on the non-critical task. Pages $P_{1,1}$, $P_{1,2}$, $P_{1,3}$ are hot memory pages for jobs in τ_1, however $P_{1,2}$ is not accessed during the considered time window.

We will use W_l to identify the minimum number of cache ways that are needed to hold the considered hot pages. Once this parameter has been calculated (see Equation 3), the first W_l cache ways are locked at system initialization. In our example $W_l = 2$, as shown in Figure 2e.

Using the legacy model (i.e. no property is enforced on the cache behavior) shown in Figure 2a, the status of the shared cache is unknown because: (1) $P_{1,1}$ and $P_{1,3}$ can evict each other, since they have the same color if no reassignment is made; (2) the non-critical task can evict any of the cache lines at any moment. Thereby, in the WCET computation we have to consider the case in which every hot page access causes cache misses. The status of the hot pages before any color/way assignment operation is shown in Figure 2c.

In the proposed model (Figure 2b), we execute a color/way assignment phase right after the start-up phase s_1. The resulting cache coordinates in terms of color and way are shown in Figure 2d. The assignment phase has a variable length that depends on the number of modified colors, but since it is executed before the first job is released, it has no impact on system schedulability. In our example, only page $P_{1,3}$ need to be recolored.

C. Cache color/way assignment

Let K be the number of available colors. In this phase to each memory page $P_{i,k} \in M_i$ $(1 \leq k \leq m_i)$ of every task τ_i is associated a couple $\langle W_{i,k}, K_{i,k} \rangle$. Here, the value $1 \leq W_{i,k} \leq W_l$ encodes the way in which the page $P_{i,k}$ will be prefetched and locked, while $1 \leq K_{i,k} \leq K$ encodes the color assigned to the page $P_{i,k}$. The assignment is done with just one constraint, namely that different way numbers shall be assigned to pages having the same color. The following property holds:

$$\begin{aligned} &\forall P_{i,k}, \forall P_{j,l} \qquad 1 \leq i,j \leq N, 1 \leq k \leq m_i, 1 \leq l \leq m_j \\ &W_{i,k} = W_{j,l} \Rightarrow K_{i,k} \neq K_{j,l} \end{aligned} \tag{1}$$

As we have previously mentioned, page coloring is a software technique which can be operated at the OS level to control the mapping between physical memory pages and cache blocks. To understand how this can be done, refer to Figure 3. Figure 3a shows the structure of a physical address from the point of view of the cache controller, as divided in tag, index, and offset. Figure 3b shows the same physical address from the point of view of the OS in a system which uses virtual-to-physical address translation.

We define the *color* of a cache page as the value of the (physical) address bits $[I: 12]$, where I is the most significant bit of the cache index (while the 12 less significant bits encode the offset in a 4 KB page). Color bits (highlighted in the

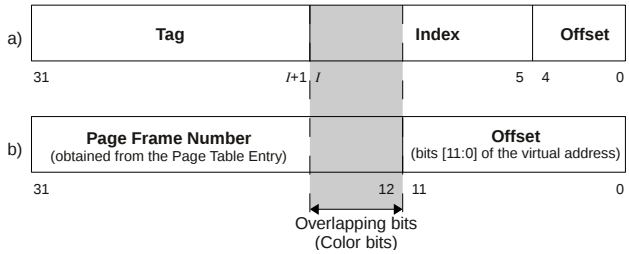

Fig. 3. Physical address structure as seen by the cache controller (a) and the OS (b).

gray area) have the following properties: (1) given the virtual address of a memory page, by manipulating color bits stored in the corresponding *Page Table Entry* the OS can influence the mapping of memory to cache page; (2) the operation is completely transparent to the task running in user space.

In an W-way set associative cache having size S_c bytes, with a page size of P_s bytes, the number of available colors K is given by:

$$K = \frac{S_c/W}{P_s} \tag{2}$$

Given K, it is possible to determine W_l for a task set:

$$W_l = \left\lceil \frac{1}{K} \sum_{i=1}^{N} m_i \right\rceil \tag{3}$$

If the way size in bytes is greater than the size of one memory page, the value of I can be calculated as:

$$I = log_2 \left(\frac{S_c}{W} \right) - 1 \tag{4}$$

In the example shown in Figure 2, the number of available colors is $K = 2$. Then, since the number of ways is $W = 2$, no more that two pages can have the same color. In the example, $P_{1,3}$ has been recolored from color 1 to color 2 (Figure 2c and 2d). In our example, $P_{1,3}$ has been assigned to way number 2, and this information will be used later to determine where to prefetch $P_{1,3}$.

D. Dynamic lockdown

During this phase we execute the prefetch and lock procedure on the given set of hot pages. As previously stated, this step can be executed at different times, depending on when the colored hot pages are requested. In our example in Figure 2, page $P_{1,1}$ is prefetched right at the end of the previous phase, while $P_{1,3}$ is prefetched at the beginning of job $\tau_{1,2}$.

As previously stated, an invariant of the procedure is that before and after the routine is executed, the first W_l cache ways are locked for every CPU in the system, so that none of them can perform an allocation in said ways.

Since lockdown by master lets us define a different cache way lockdown status for each CPU in the system, in order to prefetch and lock a given page $P_{i,k}$ on a given CPU, say CPU h, we operate as follows: (1) we make sure that none of the lines in $P_{i,k}$ are cached in any level of cache[2]; (2) we lock all the ways but $W_{i,k}$ for the CPU h; (3) we sequentially read the page $P_{i,k}$, in a non preemptive manner, in order to trigger the desired cache allocations; (4) we restore the lock status of all the W cache ways for the CPU h as it was before step (2). Once this step has been executed for a given page, we flag it as *prefetched*, so that we never run this procedure twice on the same page.

If the previous assignment constraints are met, every time a prefetch and lock procedure is executed, the following assertions hold: (1) After the invalidation and trashing step, the data contained in $P_{i,k}$ cannot be cached in a higher level of cache; in addition, it can not be cached in any of the last $W - W_l$ ways, nor in any of the first W_l ways, because they have been invalidated and locked at the very start-up of the system. (2) Every line addressed in $P_{i,k}$ will cause a cache allocation in exactly the way $W_{i,k}$, since any other way is locked for the CPU which is running the prefetch. (3) Inside the given way $W_{i,k}$ the prefetch procedure will not evict any previously prefetched page because page $P_{i,k}$ has a color $K_{i,k}$ which no other page locked on the same way can have (See Equation 1).

Therefore, if the prefetch and lockdown procedure has been executed for a given page $P_{i,k}$, then every access of a job in task τ_i to any cell contained in $P_{i,k}$ will certainly result in a cache hit. This property can be exploited in static analysis in order to compute tight WCET bounds for the critical tasks.

Note that a part of the cache will be unusable by non-critical tasks, and this can have a negative impact on their average performances. However, this is usually an acceptable tradeoff in real-time systems. Note also that a subset of hot pages could be shared among a group of tasks. Nonetheless, handling a similar case requires additional logic and it goes beyond the scope of this work.

Finally, based on experimental results, it is worth to mention that task execution time as a function of the number of locked pages is well approximated by a convex function (see Figure 5 and 6). This has an important consequence; in fact, whenever the total number of hot pages (in the task set) exceeds the available cache size, an optimal cache allocation to multiple tasks can be approximately computed by using a convex optimization algorithm [33] like Q-RAM [34].

VI. Evaluation

In this section we present the results obtained with an experimental implementation of the proposed framework, as well as the relevant hardware details of the testbed platform.

A. Methodology

We adopted for this work a Pandaboard development board. It is a low-cost embedded ARM platform featuring a OMAP4430 processor [35]. This processor incorporates a dual-core ARM Cortex-A9 MPCore with symmetric multiprocessing (SMP). The OMAP4430 operates at a clock frequency of 1.0 GHz. The total DRAM available is 1GB and there are two cache levels [36]. In particular, there are two L1 caches which are private to each core. Since our interest is to study the interference on the shared cache, the L1 caches have been disabled to perform the experiments. In general, performing the evaluations on the last level (shared) cache disabling the higher levels allows us to obtain an upper bound of the response time of the cache hierarchy. Such upper bound can be more or less pessimistic according to the cache features and the memory footprint of the tasks under analysis.

[2]This step is hardware specific. It can be done by trashing higher cache levels and, on the last level, through by-line invalidation mechanisms (if supported), or invalidating all the last $W - W_l$ cache ways. Doing so, no line in the page that is going to be prefetched could trigger a cache hit in any other cache level or way, thereby causing an allocation in exactly way $W_{i,k}$.

The second and last level of cache is a unified, physically indexed, physically tagged L2 cache. It is shared among the two Cortex-A9 cores, has a total size of 1024 KB and it is internally organized as a 16-way set associative cache. The cache is controlled by a hardware circuit called PL310 which exposes a set of memory mapped registers to control the cache behavior [32]. The PL310 supports the following lockdown policies: by line, by way, by master. The lockdown by master policy works as we have explained before. Lockdown by way is the lockdown by master policy when a single CPU is connected to the cache controller. Finally, the provided lockdown by line mechanism is not atomic, thus implementing Colored Lockdown using such lockdown policy is non trivial.

Each Cortex-A9 core features a set of performance counters, including a 32-bit clock cycle counter. We have used this counter to collect accurate execution time measurements during our experiments because it can be configured to be accessed from user space with a negligible overhead.

For our experiments, we have used a set of benchmarks from the EEMBC AutoBench suite [37] which are reported in Table II. The benchmarks in this suite feature algorithms to test the performance of embedded platforms in automotive and industrial applications. For this reason, they share some key characteristics with real-time applications (limited memory footprint, limited use of dynamic memory, etc.) and can therefore be considered representative of a typical real-time workload.

As reported in [37], each benchmark features an iterative structure and allows us to specify a customized number of iterations per each run. However, since the execution time of each iteration is too small to be considered a job of a typical real-time task, we performed some adaptations to make each adopted benchmark compliant with the periodic task model. In particular, we have determined the number I of iterations, reported in the last column of Table II, needed by the algorithm to loop over the input buffer once. In this way, we consider a job as the aggregation of I iterations. Using a combination of `setitimer`/`sigaction` system calls, a new job is released every 30 ms. Each sample presented in the following results summarizes the execution of 110 jobs. The first 10 jobs are used to put the cache in a *hot* status, therefore their execution time is not accounted.

Moreover, to simulate a partitioned scheduler, we bind the benchmark tasks to the first CPU and schedule them according to a real-time, fixed-priority scheduling policy. Interference on the shared cache is created running a synthetic, memory intensive task on the second CPU.

B. Memory Profiling

The results of running the memory profiler on the seven EEMBC benchmarks can be seen in Table II. The second column displays how many different memory pages were found to be accessed by a given benchmark. This number is equivalent to the number of lines in the profile file, similar to the sample one in Figure 1f. In our experiments, we have chosen to color/lock a subset of the resulting pages. In particular, the third column of Table II shows how many pages per each benchmark have been selected. The fourth column reports the percentage of accesses that fall in the selected pages with respect to the total number of accesses. As it is shown in the table, we have selected the minimum number of pages so to cover at least the 80% of accesses.

Benchmark	Total pages	Hot pages	% accesses in hot pages	Iters/Job
a2time	15	4	81%	167
basefp	21	6	97%	675
bitmnp	19	5	80%	43
cacheb	30	5	92%	22
canrdr	16	3	91%	498
rspeed	14	4	85%	167
tblook	17	3	81%	78

TABLE II
MEMORY PROFILING RESULTS

C. Colored Lockdown Results

Once we generated the memory profile for each benchmark, we compared the execution time of these benchmarks with and without Colored Lockdown. The results of these experiments can be seen in Figure 4. The plotted values are those of the observed worst case. The first bar displays the execution time of the benchmark without any protection and in isolation. The second bar displays the execution time of the same task still without protection, but, this time, with the interference task running on the second CPU. In all the benchmarks, in the latter case, the execution time increases significantly, sometimes more than 2.5 times. Finally, the third bar of each cluster displays the observed worst case execution time when interference is still present and the Colored Lockdown protection is enforced. All bars are normalized to the first bar. We observe that, in most cases, Colored Lockdown eliminates the effects of the interference task, leading us to the conclusion that Colored Lockdown effectively isolates the task by eliminating the interference on the shared last level cache. In some cases, enforcing protection through Colored Lockdown slightly improves execution time with respect to the isolation case. This is due to the fact that self-evictions (cache evictions caused by the same task) are also reduced.

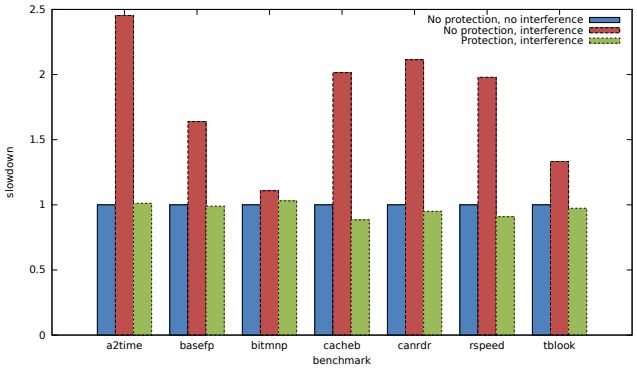

Fig. 4. Graph of observed worst case execution time for each benchmark, with 1) no interference and no protection, 2) interference and no protection, and 3) interference and protection.

The number of pages locked by Colored Lockdown in the above mentioned results were also minimized: between three to six pages were locked in each benchmark. For instance, in the case of a2time, we see from Table II that as many as 15 pages were found to be accessed by the benchmark, but only the four hottest pages needed to be locked to significantly reduce the stall time caused by cache interference. The graph in Figure 5 displays the average execution time as a function of how many pages were locked. The horizontal line

displays the execution time of the benchmark under normal circumstances (no interference task on the second CPU and no Colored Lockdown protection). One can observe that the benefit of locking more pages diminishes after 4 pages, which corresponds to the case where 81% of memory accesses are locked in the last level cache and do not require a DRAM access. In a scenario where the total memory footprint is bigger than the cache size, this type of analysis helps selecting those pages which provide the highest benefit if locked in cache. A similar effect can be seen in the canrdr benchmark, in Figure 6, where similarly only 3 pages need to be locked. Similar curves, that we are not showing for sake of space, have been obtained for all the benchmarks. In both the presented figures, the depicted error bars represent the observed best and worst execution time over the 100 measurements.

Note that through profiling we obtain a correct rank of memory pages according to how frequently they are accessed. Conversely, if the task behavior is not inspected through profiling, it is hard to find an efficient solution to determine such curves, because it would involve running the task several times.

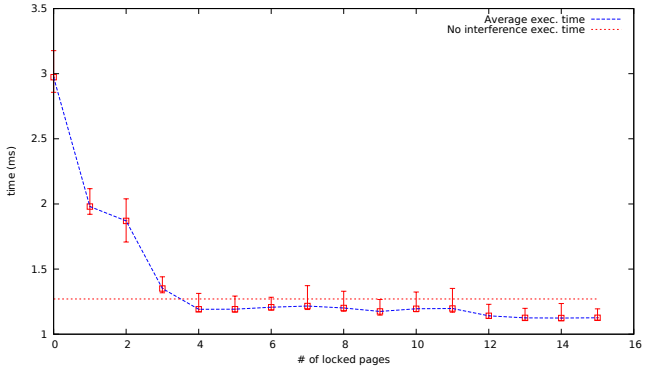

Fig. 5. Graph of execution time depending on the number of pages locked, for the benchmark a2time.

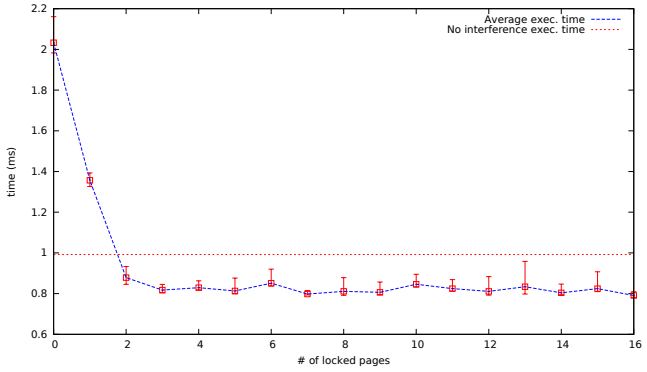

Fig. 6. Graph of execution time depending on the number of pages locked, for the benchmark canrdr.

Another experiment has also been carried out to understand the behavior of multiple tasks with different priorities scheduled at the same time on the same CPU. Again, we are interested in comparing what happens if protection is enforced using Colored Lockdown versus the case in which no cache allocation is performed. The results are shown in Figure 7. In the graph, the first four bars represent the normalization base for the following cases and correspond to the case in which all the tasks are running without protection and no interference is generated on the last level cache by the other core. Note that, in this case, some cache interference coming from the other three tasks on the same CPU is suffered by each task. However, since the footprint of the considered tasks is small, applying the protection in this case does not influences performances much, as shown by the second group of bars.

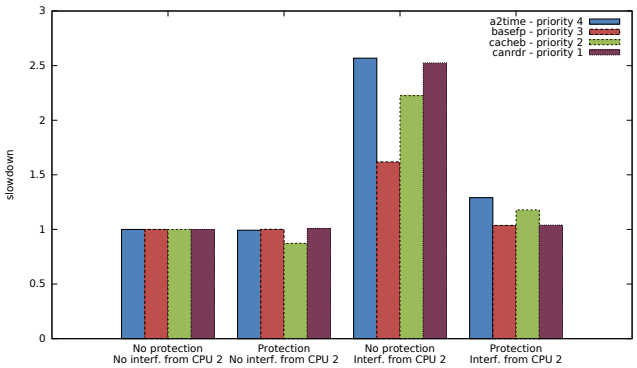

Fig. 7. Resulting observed average time for four benchmarks scheduled at the same time with different priorities. The highest priority level (a2time) is the top priority.

According to the depicted values, in case interference in the last level cache is introduced running memory intensive processes on the second CPU (third group of bars), even the highest priority task can suffer a 2.5x slowdown. Conversely, in the last cluster of bars, we show the behavior of the system when, per each task, the number of hottest memory pages reported in Table II is colored and locked in the last level cache. In this case, the enforced protection enhances the isolation between all the tasks of the system, so that just a negligible slowdown is suffered. To understand the source of this slowdown, we can compare Figure 4 and Figure 7. In particular, considering the cases where interference is generated on CPU 2 and no protection is enforced (Figure 4, second bar of each cluster), we note that almost all the tasks run slower in the co-scheduling experiment (Figure 7, third cluster). This is because, in the latter case, DRAM accesses generated by inter-task interference on non-locked pages compete with those generated by CPU 2. In fact, this effect is not visible when CPU 2 is idle (in the first and second cluster of Figure 7) but generates the slight slowdown observed in the last cluster of the same result set.

VII. Conclusion

Memory resources are the main bottleneck in many real-time multi-core systems. In this work, we have proposed a complete cache management framework for real-time systems which integrates a memory analysis tool and a technique to perform deterministic cache allocation.

The memory analysis tool leverages on profiling techniques to analyze the memory access pattern of a given task and to determine the most frequently accessed memory pages.

Moreover, colored lockdown is used to manage the content of a cache shared among two or more tasks, both in single-core and multi-core platforms. This technique combines page coloring and cache lockdown in order to enforce a deterministic cache hit rate on a set of hot memory regions.

Exploiting the presented framework, it is possible to increase isolation among tasks, a key property for certifiable hard real-time systems. In fact, it can be used to avoid critical tasks from suffering interference coming from self-eviction, preemption, asynchronous kernel flows (i.e. ISRs), and other tasks running on the same core or on a different one.

We have fully implemented the proposed techniques on a commercial multi-core embedded platform and we have shown that enforcing such protection can sharply improve the schedulability of critical tasks.

As a part of our future work, we plan: (1) to extend the implementation of our profiling technique to merge in a single profile the data collected during multiple executions of the same task with different input vectors; (2) to correlate the data about memory accesses with timing information, in order to detect a change in the memory workload of a task across time and adapt the Colored Lockdown strategy accordingly.

REFERENCES

[1] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. *ACM Trans. Archit. Code Optim.*, 8(4):34:1–34:25, January 2012. ISSN 1544-3566.

[2] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008. ISSN 1539-9087.

[3] Heiko Falk and Helena Kotthaus. WCET-driven cache-aware code positioning. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 145–154, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0713-0.

[4] Tiantian Liu, Minming Li, and Chun Jason Xue. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Syst.*, 48(2):166–197, March 2012. ISSN 0922-6443.

[5] Frank Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18:217–247, May 2000. ISSN 0922-6443.

[6] J.R. Larus. Efficient program tracing. *Computer*, 26(5):52 –61, may 1993. ISSN 0018-9162.

[7] Pin - a dynamic binary instrumentation tool. URL `http://www.pintool.org/`.

[8] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *In Proceedings of the 2007 Programming Language Design and Implementation Conference*, 2007.

[9] Quan Sun and Hui Tian. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, Galveston Island, Texas, USA, October 2011.

[10] Chris Lattner and Vikram Adve. LLVM - a compilation framework for lifelong program analysis & transformation, 2004.

[11] Quan Sun and Hui Tian. A flexible automatic source-level instrumentation framework for dynamic program analysis. In *Software Engineering and Service Science (ICSESS), 2011 IEEE 2nd International Conference on*, pages 401 –404, july 2011.

[12] Bruce Jacob, Spencer Ng, and David Wang. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann, 2007. ISBN 9780123797513.

[13] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pages 300–303, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6.

[14] Andrew Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2:315–327, March 1994. ISSN 1069-5451.

[15] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33:67–74, December 2000. ISSN 0018-9162.

[16] Jorg Herter, Peter Backes, Florian Haupenthal, and Jan Reineke. CAMA: A predictable cache-aware memory allocator. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, pages 23–32, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4442-7.

[17] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30:125–133, November 1995. ISSN 0362-1340.

[18] D. B. Kirk and Jay K. Strosnider. SMART (Strategic Memory Allocation for Real-Time) cache design using the MIPS R3000. In *IEEE Real-Time Systems Symposium '90*, pages 322–330, 1990.

[19] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 14:1–14:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8.

[20] Ravi Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, ICS '04, pages 257–266, New York, NY, USA, 2004. ACM. ISBN 1-58113-839-3.

[21] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 114–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1851-6.

[22] E. Tamura and Javeriana Cali. Towards predictable, high-performance memory hierarchies in fixed-priority preemptive multitasking real-time systems, 2007.

[23] E. Tamura, F. Rodrguez, J. V. Busquets-mataix, and A. Mart Campoy. High performance memory architectures with dynamic locking cache for real-time systems. Technical report, Department of Computer Science and Engineering. University of Nebraska-Lincoln, 2004.

[24] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez, and Víctor Viñals. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *J. Syst. Archit.*, 57: 695–706, August 2011. ISSN 1383-7621.

[25] Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. OS-Controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, pages 213–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8016-4.

[26] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 367 –378, feb. 2008.

[27] L. Soares, D. Tam, and M. Stumm. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 258 –269, nov. 2008.

[28] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9.

[29] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating System and Computer Architecture*, June 2007.

[30] Junghoon Kim, Inhyuk Kim, and Young Ik Eom. Code-based cache partitioning for improving hardware cache performance. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 42:1–42:5, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1172-4.

[31] I. Puaut. Real-time performance of dynamic memory allocation algorithms. In *Real-Time Systems, 2002. Proceedings. 14th Euromicro Conference on*, pages 41 – 49, 2002.

[32] PrimeCell Level 2 Cache Controller (PL310) - technical reference manual.

[33] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521833787.

[34] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, RTSS '97, pages 298–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8268-X.

[35] OMAP4430 Multimedia Device - rev 2.0 - technical reference manual.

[36] ARM Cortex-A9 - technical reference manual.

[37] EEMBC AutoBench v1.1 - data book.