A Survey on Cache Management Mechanisms for Real-Time Embedded Systems

GIOVANI GRACIOLI, Federal University of Santa Catarina
AHMED ALHAMMAD, University of Waterloo
RENATO MANCUSO, University of Illinois at Urbana-Champaign
ANTÔNIO AUGUSTO FRÖHLICH, Federal University of Santa Catarina
RODOLFO PELLIZZONI, University of Waterloo

Multicore processors are being extensively used by real-time systems, mainly because of their demand for increased computing power. However, multicore processors have shared resources that affect the predictability of real-time systems, which is the key to correctly estimate the worst-case execution time of tasks. One of the main factors for unpredictability in a multicore processor is the cache memory hierarchy. Recently, many research works have proposed different techniques to deal with caches in multicore processors in the context of real-time systems. Nevertheless, a review and categorization of these techniques is still an open topic and would be very useful for the real-time community. In this article, we present a survey of cache management techniques for real-time embedded systems, from the first studies of the field in 1990 up to the latest research published in 2014. We categorize the main research works and provide a detailed comparison in terms of similarities and differences. We also identify key challenges and discuss future research directions.

Categories and Subject Descriptors: C.5.3 [**Computer systems organization**]: Real-time systems; E.1.2 [**Software and its engineering**]: Operating systems; E.2.4 [**Software and its engineering**]: Compilers

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Real-time systems, cache partitioning, cache locking, memory allocators, compilers

1. INTRODUCTION

Real-time embedded systems are nowadays present in virtually any environment. Areas such as automotive, avionics, and telecommunications are examples where real-time embedded systems can be vastly found. In a *real-time system*, the correctness of the system depends not only on its logical behavior, but also on the time in which the computation is performed [Liu and Layland 1973]. The main distinction is between *soft real-time* (SRT) and *hard real-time* (HRT) systems. In both, applications are typically realized as a collection of *real-time tasks* associated with timing constraints and scheduled according to a chosen *scheduling algorithm*. However, in an HRT, the loss of a time constraints (*i.e.,* a *deadline*) may cause uncountable or catastrophic damage, such as human lives or a considerable amount of money. In a SRT, instead, missing a deadline results in just a degradation of the qualify of service (QoS) provided by

Author's addresses: G. Gracioli, Center for Mobility Engineering, Federal University of Santa Catarina, Brazil; A. A. Fröhlich, Department of Computer Science, Federal University of Santa Catarina, Brazil; A. Alhammad and R. Pellizzoni, Department of Electrical and Computer Engineering, University of Waterloo, Canada; R. Mancuso, Department of Computer Science, University of Illinois at Urbana-Champaign, USA.

DOI:http://dx.doi.org/10.1145/0000000.0000000

the system. Hence, in order to provide a correct behavior and a good QoS, real-time embedded applications must be designed to always meet their deadlines.

At the same time, the continuous evolution of processor technology, together with its decreasing cost, has enabled multicore architectures (*e.g., Symmetric Multiprocessing - SMP*) to be also used in the real-time embedded system domain. Besides, real-time embedded applications are demanding more processing power due to the evolution and integration of features that can only be satisfied, in a cost-effective way, by the use of a multicore platform. In an automotive environment, for instance, new safety functionalities like "automatic emergency breaking" and "night view assist" must read and fuse data from sensors, process video streams, and rise warnings when an obstacle is detected on the road under real-time constraints [Mohan et al. 2011]. This is a typical scenario where an increasing demand for advanced features results in a proportional demand for additional computational power. Moreover, an increase of system functionalities determines additional costs in terms of power consumption, heat dissipation, and space occupation (*e.g.,* wiring) [Cullmann et al. 2010]. Thereby, multicore processors represent a cost-effective solution to decrease the mentioned costs, since the additional computational demand can be allocated on a single processing unit, instead of several processing units spread over the vehicle.

In general, the capability of a real-time system to meet its deadline is verified through a *schedulability analysis* [Davis and Burns 2011]. A basic assumption, which is common to all such schedulability analysis techniques, is that an upper bound on the *Worst-Case Execution Time* (WCET) of each task is known. However, deriving safe yet tight bounds on task WCET is becoming increasingly difficult. This is especially true on multicore architectures, because processors (or cores) share hardware resources, such as cache memory hierarchy, buses, DRAM and I/O peripherals. Therefore, operations performed by one processing unit can result in unregulated contention at the level of any shared resource and thus unpredictably delay the execution of a task running on a different core. One of the main factors for unpredictability, which is also the focus of this survey, is the *CPU cache memory hierarchy* [Suhendra and Mitra 2008; Muralidhara et al. 2010; Zhuravlev et al. 2012]. CPU caches of a modern multicore typically consist of two or three levels of cache memory placed between the core and the main memory to bridge the gap on the high processor speed and low memory performance. Usually, a smaller Level-1 (L1) cache is private to each core, while a larger Level-2 (L2) and/or a Level-3 (L3) cache is shared among all the cores or a cluster of cores. The last level of cache before the main memory, is usually referred as Last Level Cache (LLC). In case of a *cache miss*, the requested instruction or data must be brought from the higher levels of cache or from the main memory, occurring in larger execution times.

CPU caches are hardware components that are mostly transparent to the programmer and that rely on temporal and space locality of memory accesses to reduce the average execution time of applications. Thereby, they employ a set of heuristics to keep data that are more likely to be accessed in a near future and displace old, non-referenced entries. At a high level, the heuristic behavior of a cache means that a memory access in the same location throughout the execution of a task, may or may not result in a *cache hit*, depending on the history of the system. In fact, as it will be clear in Section 2.1, the execution pattern of the task itself, of different tasks on the same core or even of a set of tasks on a different core can impact the *cache hit ratio* of a given memory access pattern. This means that a complex function of the system history can directly impact the time required by a task to retrieve data from the memory hierarchy, explaining why cache memories are one of the main sources of unpredictability. Also, the memory access behavior of a real-time application is driven by its functionalities. For instance, simple HRT control loops do not demand a high mem-

ory bandwidth, while video processing applications demand both memory bandwith and latency.

In this work, our focus is on techniques that address conflicts in the allocation of cache lines (spatial contention). However, as multiple cores perform contemporary accesses to the same cache level, contention on the temporal domain can also occur. Although temporal contention can produce non-negligible effects, we only set our focus on spatial contention for two main reasons. First, because the magnitude of time unpredictability that arises from spatial contention is typically higher than what observable due to temporal contention. This is generally true for embedded multicore systems that are not classified as many-core systems (*i.e.,* up to 16 or 32 cores that do not use complex on-chip networks). Second, because the problem of managing temporal contention at the cache level is not significantly different than DRAM bus management. To cope with a space contention problem, CPU caches need to be directly or indirectly managed according to an allocation scheme. Allocation schemes can exhibit different properties, applicability, and can be more or less effective in protecting the timing behavior of real-time tasks (see Section 2).

1.1. Cache Interferences

The execution time of a real-time task in a multicore processor[1] can be affected by a number of different types of interference, depending on the behavior of the cache hierarchy:

— **Intra-task interference:** intra-task interference occurs when tasks have their working set sizes larger than a specific cache level, or, in general, when two memory entries in the working set are mapped in the same cache set. The consequence in this case is that a task evicts its own cache lines. Intra-task interference also happens in single-core systems.
— **Intra-core interference:** intra-core interference happens locally in a core[2]. Specifically, when a preempting task evicts the preempted task's cached data. As a result, the preempted task will experience an increase in its data access time (and thus a delay) as soon as it is rescheduled. The severity of the experienced delay depends on the particular cache line replacement policy implemented by the cache, as well as the length of the preemption and the data access pattern of the preempting task [**?**; **?**; Reineke et al. 2007; Grund and Reineke 2010a; Kim et al. 2013].
— **Inter-core interference:** inter-core interference is present when tasks running on different cores concurrently access a shared level of cache [Kim et al. 2013]. When this happens, if two lines in the two addressing spaces of the running tasks map to the same cache line, said tasks can repeatedly evict each other in cache, leading to complex timing interactions and thus unpredictability.
  Inter-core interference can also occur due to Simultaneous Multi-Threading (SMT), when two or more hardware threads share the same private cache levels (L1 and/or L2). Since this type of interference is suffered by tasks that can run in parallel, an exact analysis requires analyzing all the possible interleaving of task executions. This combinatorial problem is clearly intractable, thus, inter-core interference results extremely difficult to integrate into a static analysis framework [Guan et al. 2009].

Furthermore, memory accesses originating from a core can be classified as either demand accesses, if required by instructions executed on the core, or prefetch accesses, if

[1]In the rest of the text, we will use the term 'processor' referring to the physical chip and the term 'core' to refer to each single processing element in a multicore chip.
[2]Intra-core interference also happens in single-core processors and it is referred as cache-related preemption delay.

speculatively issued by a hardware prefetcher unit. Although important for the average performance of the system, it is common practice (in real-time systems) to disable hardware prefetchers to eliminate prefetch interference, thus making the processor more predictable. Hence, in this survey, we assume that prefetchers are disabled and do not cause additional interference. In Section 7, we discuss open issues and future directions considering also the usage of prefetchers in real-time systems.

1.2. WCET Derivation

The main objective of cache management schemes for real-time systems is to deal with the described problems, simplifying the estimation of the tasks' WCET. WCET estimation typically follows one of two main approaches: static analysis or empirical measurement [Wilhelm et al. 2008]. In the former approach, a tool attempts to provide the WCET by analyzing the application binary code without executing it directly on the hardware. Generally, WCET estimation tools are available only for simple processors, due to sophisticated hardware features, such as caches, branch predictors and pipelines, which make the static analysis extremely difficult or overly pessimistic [Wilhelm et al. 2008]. In the latter approach, the application binary code is executed on the hardware platform for multiple times, and an estimation of the WCET with an adjustable confidence is extracted from these executions. Moreover, in measurement-based approaches, the resulting value of WCET is typically inflated further with an error margin (20% to 30% the observed value) in order to account for unobserved conditions that can delay the execution. Consequently, measurement-based approaches can also lead to overestimated WCET.

The use of caches makes both static and measurement-based WCET estimation more complex, because the execution time of an instruction may vary depending on: (A) the data/instruction location in the memory hierarchy; (B) if a memory access results in a miss or a hit in any of the cache levels; (C) which cache coherence protocol is being used in case of true or false sharing; and (D) which cache replacement policy is implemented on the cache controller [Zhuravlev et al. 2012]. Note that while several real-time cache analysis frameworks have been proposed for single-core systems [Wilhelm et al. 2008], current static analysis methodologies provide pessimistic bounds on cache misses for shared caches. Furthermore, to the best of our knowledge, no existing static analysis technique is able to account for the coherence protocol effects.

Note that, even if WCET estimation is possible, the presence of inter-core interference greatly complicates validation and certification of multicore real-time systems. Allowing hardware components of commodity multicore platforms to operate in an unrestricted manner makes it significantly harder to provide temporal and spatial isolation for systems that demand such isolation (avionics for instance). In fact, if no predictable arbitration mechanism for shared resources is in place, the pessimism on task WCET can easily result to be unacceptable. This problem has been recognized by the Certification Authorities Software Team (CAST), an international group of avionics certification and regulatory representatives from North and South America, Europe, and Asia [Certification Authorities Software Team (CAST) 2014].

1.3. Existing Solutions and Contributions

Several works have been proposed to cope with the described cache memory hierarchy problems in the context of real-time embedded systems and to tighten the WCET estimation. In general, such works rely on enforcing timing isolation between sub-components in the system: the execution time of one sub-component should not depend on the behavior of other sub-components. Such isolation can be implemented (A) at the level of a single task, where sub-components are functions or basic blocks in the task's code; (B) at the level of a core, where sub-components are tasks; or (C) at the multi-

core system level, where sub-components are individual cores or software partitions running on each core.

Two approaches are most commonly used to enforce a more deterministic behavior on CPU caches. The first approach to timing isolation is cache partitioning, which divides the cache in partitions and assigns specific partitions to tasks or cores [Liedtke et al. 1997; Mancuso et al. 2013; Ward et al. 2013; Kim et al. 2013]. Based on the structure of a set-associative cache, these approaches can be further categorized in: (A) index-based cache partitioning, when partitions are formed as an aggregation of associative sets in the cache (horizontal slicing); and (B) way-based cache partitioning, when each available cache way is exclusively assigned to a single partition.

The second technique that has been investigated is cache locking. Cache locking relies on hardware features present in many embedded platforms. Specifically, it is possible to flag a given cache line or way as locked, thus preventing its content from being evicted until a successive unlock operation is performed [Aparicio et al. 2011; Sarkar et al. 2012; Mancuso et al. 2013].

Both cache partitioning and locking can be available to applications through the operating system (OS) memory allocator. The problem of developing OS-level memory allocators that can perform cache-aware allocations and feature a predictable execution time has been addressed in [Chilimbi et al. 2000; Afek et al. 2011; Herter et al. 2011]. The common denominator for all the aforementioned techniques is to improve the predictability of real-time systems deployed on top of cache-based architectures, in order to provide better isolation guarantees for real-time embedded applications. Even though the focus of this survey is real-time systems, we also include some related work for general purpose systems. We note that the optimization techniques for the latter systems are throughput oriented as opposed to worst-case oriented. In the process of improving the overall throughput, the timing requirement of each task is not guaranteed to be met. In contrast, real-time tasks care about meeting their timing requirements, and there is no gain in improving above such requirement. Still, some of these average-case optimization techniques can be adapted for SRT systems.

To the best of our knowledge, no complete classification of these works has been attempted so far. Aiming at closing this gap, this article presents a survey of cache management mechanisms for real-time embedded systems, from the first studies of the field in 1990 up to the latest research published in 2015. The objective of the survey is to categorize each approach and to provide a detailed comparison in terms of similarities and differences. In summary, our contributions are:

— We classify each related work into one of the following categories: index-based cache partitioning, way-based cache partitioning, cache locking, or OS memory allocators. We also mention which techniques exploits more than one mechanism to improve predictability.
— For each category, we present an overview of the approach (index-based and way-based cache partitioning, cache locking, and memory allocators) and discuss the main characteristics of each work. In particular, we detail: (A) whether the approach is implemented in hardware or software, and in the latter case, whether it requires compiler or OS support; (B) what type of isolation it provides, i.e., whether it addresses intra-task, intra-core, or inter-core interference; (C) the level of isolation, i.e., how effective the approach is at preventing the addressed source of interference; (D) any further limitation or key assumptions in the work that might limit its applicability, in particular in an industrial context.
— We provide a discussion on open issues and future directions in the field.

The remainder of this article is organized as follows: Section 2 provides background on current processor and memory organizations. It describes the problems raised by

these organizations. The following four sections present our categorization of related works. Section 3 presents all related works categorized as index-based cache partitioning. Section 4 categorizes the related work in way-based cache partitioning and Section 5 as cache locking mechanisms. Section 6 presents operating systems memory allocators that either use one of the three previous techniques or are designed to be predictable. In the end of these four section, we present a table summarizing the main characteristics of each work. Section 7 identifies open problems and future directions in the field. Finally, Section 8 concludes the article.

2. PROBLEM DESCRIPTION AND BACKGROUND

In this section we briefly introduce background concepts needed to follow the rest of the paper and present our categorization of cache management mechanisms real-time embedded systems.

2.1. Cache Line Replacement Policies

Ideally, all data should be available on the caches, thus improving the overall program execution time. As caches have a limited space, the problem is to keep in the cache only the most important data for a given window of time. A cache line replacement algorithm is responsible for choosing which cache line is replaced when a cache miss occurs. The cache line replacement policy depends on the cache organization (*i.e.,* cache mapping). Currently, the most used cache organization is a **set associative cache**. In a set-associative cache, a memory block has a fixed number of positions in which it can be placed in the cache. An n-way set-associative cache has "n" locations for a memory block. Also, the cache consists of a number of sets, each of which consists of "n" blocks. A main memory block maps to a unique set in the cache and the block can be placed in any element of that set.

Least Recently Used (LRU), First In First Out (FIFO), and Pseudo-LRU (PLRU) are examples of cache replacement algorithms currently used by multicore processors. For instance, the TRICORE 1798, several POWERPC variants (MPC603E, MPC755, MPC7448) [Grund and Reineke 2010a], and Intel Pentium II-IV [Reineke et al. 2007] use PLRU. Intel XScale, ARM9, and ARM11 use FIFO [Reineke et al. 2007]. Intel Pentium I and MIPS 24K/34K use LRU [Reineke et al. 2007].

2.2. Virtual Memory

The key idea behind virtual memory is to divide the address space of a program in blocks, called *pages*. Each page is a series of contiguous memory addresses. Pages are mapped into physical memory locations, but do not need to be in the physical memory to execute a program. The Memory Management Unit (MMU) translates logical addresses in a page to physical memory addresses dynamically as programs access their own pages. This mechanism is called *paging* and it is available on the majority of systems that support virtual memory. Paging is the key to perform some cache partitioning mechanisms, as will be discussed in Section 3.

2.3. Cache Coherence Protocols

Each core has its own data and uses its private data cache for speeding up the processing. However, when cores share data, each copy of the data is placed in the core's private cache and the cache coherence protocol is responsible for keeping the consistency between each copy (usually through bus snooping[3]). Whenever a core writes into

[3]Although important directory-based protocols are not discussed in this paper, because they are usually implemented in non-uniform memory access (NUMA) processors. Such processors are not preferable for real-time embedded systems due to their unpredictability.

a data that other cores have cached, an invalidation occurs, increasing tasks' execution time. This invalidation operation is performed automatically by the hardware and take hundreds of cycles (about the same time as accessing the off-chip RAM), increasing the application's execution time [Boyd-Wickizer et al. 2010]. Two kinds of scaling problem occur due to shared memory contention [Boyd-Wickizer et al. 2010]: access serialization to the same cache line done by the cache coherence protocol, which prevents parallel speedup, and saturation into the inter-core interconnection, also preventing parallel processing gains. MESI [Hennessy and Patterson 2006], MOESI [AMD 2013], and MESIF [Intel 2010] are the most used cache coherence protocols by current multicore processors. MESIF and MOESI protocols are usually used in ccNUMA architectures, such as Intel Sandy Bridge and AMD Opteron respectively, while MESI is widely used in UMA architectures, such as Intel dual-core and ARM Cortex-A9.

2.4. Categorization of Cache Management Mechanisms

We categorize the cache management mechanisms for real-time embedded systems in cache partitioning and cache locking. In a cache partitioning mechanism, the central idea is to assign a portion (*i.e.,* a partition) of cache to a given task or core in the system to reduce inter-core interference, to increase the predictability, and to ease WCET estimation. There are two forms of cache partitioning: index-based or way-based partitioning. In the former, partitions are formed as an aggregation of associative sets in the cache. In the latter, partitions are formed as an aggregation of individual cache ways. Figure 1 shows an example of the two cache partitioning approaches in an n-way set-associative cache. In Figure 1(a), each set is considered a different and isolated partition (horizontal slicing). One or more sets are individually assigned to a task or core and all memory allocations performed by this task or core is mapped to the assigned set(s). In Figure 1(b), each way is considered an individual partition and one or more ways can be assigned to a task or core (vertical slicing). Cache partitioning can be further divided in hardware- or software-based approaches. Moreover, software-based approaches may need a compiler or an OS support.

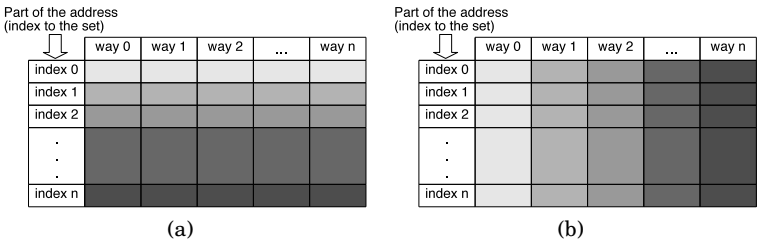

Fig. 1. Classification of cache partitioning approaches: (a) overview of the index-based partitioning. (b) overview of the way-based partitioning.

The second cache management mechanism for real-time embedded systems is cache locking. The central idea behind cache locking is to lock a portion of the cache in order to exclude the contained lines in this portion from being evicted by the cache replacement policy and by intra-task, intra-core, or inter-core interferences. Cache locking is a hardware-specific feature, which typically is done at a granularity of a single way or line. Figure 2 presents two cache locking variations. Figure 2(a) shows the locking of an entire way. The lock of a whole way means that the contents within that way across all sets cannot be evicted. Locking a whole way has not been explored deeply, because the number of ways is usually limited (in the range from 4 to 32) [Mancuso et al. 2013]. Figure 2(b) shows the locking of an individual way. The cache line locking strategies

provided by most of the current commercial embedded platforms use a set of registers to: a) mark portions of cache (typically entire ways) as locked; or b) to enable the locking of all the lines fetched after the change in the status a configuration register. These mechanisms are considered non-atomic as opposed to the support in the CPU ISA for atomic fetch-and-lock instructions. Non-atomic cache locking mechanisms make it difficult to predict what is cached and what is not. Moreover, multicore shared caches are usually physically indexed and tagged. This means that in the worst case, physical pages allocated to tasks can map to the same cache sets, and thus that no more than n locked lines can be kept at the same time, where n is the cache associativity. This problem can be overcome if explicit control is enforced on the physical addresses of the locked entries (usually by the OS), as proposed in [Mancuso et al. 2013].

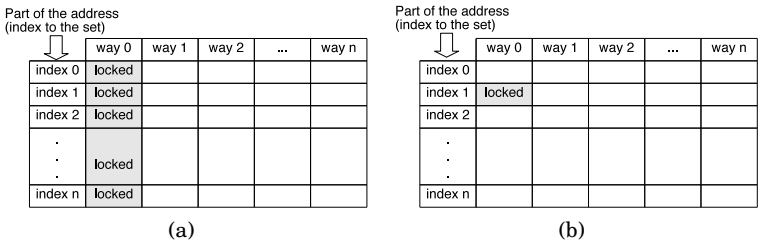

Fig. 2. Examples of ache locking variations: (a) overview of the way locking. (b) overview of the cache line locking.

Both cache locking and partitioning can be available to applications through the OS memory allocator. An OS memory allocator is responsible for managing free blocks of memory in a large pool of memory (*heap*), serving the application requests for memory spaces. Two main concerns are common to OS memory allocators: performance and fragmentation. Performance depends on the ability of the allocator to organize free blocks of memory such that the search for a block is performed efficiently. Fragmentation depends on the ability to avoid small gaps between allocated memory blocks. Besides performance and fragmentation, a memory allocator for real-time embedded systems must be predictable, i.e., the time to allocate a memory block, despite its size, must be known. Furthermore, a memory allocator can also provide means for applications to use a cache partitioning or locking mechanism in a transparent and simple way. In this survey, we provide an overview of OS memory allocators that are either predictable or cache-aware.

2.5. Cache Analysis Methods

As we mentioned early, several well-developed cache analysis techniques have been proposed for single-core processors. These techniques analyze the interference due to intra-task and intra-core cache conflicts. The latter is known as cache related preemption delay (CRPD). The CPRD focuses on cache reload overhead due to preemptions while the intra-task analysis focuses on the cache conflicts within the same task assuming non-preemptive execution. Since the focus of this paper is on techniques that manage the cache (spatial isolation rather than joint analysis), a detailed description of such analysis techniques is out of the paper's scope.

Unfortunately, the single-core timing analysis techniques are not applicable for multicore processors with shared caches. In this case, inter-core interference is caused by tasks that can run in parallel and this requires analyzing all system's tasks. The analysis of non-shared caches has been already considered as a complex process and extending it to shared caches is even harder. In fact, the researchers in the community

of WCET analysis seems to agree that "it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention between multiple cores in a shared cache" [Suhendra and Mitra 2008]. Despite this challenge, few works have been proposed to address the problem of shared caches. However, these techniques are only applicable for simple architectures and statically scheduled tasks.

The first work that studies the analysis of shared caches in multicore processors is proposed in [Yan and Zhang 2008]. This work assumes a system with two tasks simultaneously running on two cores with direct-mapped shared instruction cache. Later, a cache conflict graphs were used to capture the potential inter-core conflicts [Zhang and Jun 2012]. The work in [Liang et al. 2012] improves upon [Yan and Zhang 2008] by exploiting the lifetime information of tasks and bypassing the disjoint tasks (tasks that cannot overlap at run-time) from the analysis. This work assumes a task model where all tasks are synchronized. Clearly, for systems with dynamic scheduling, it will be extremely difficult to identify the disjoint tasks. Other research [Hardy et al. 2009] proposes to bypass the shared cache for single-usage cache lines to avoid inter-core conflicts and therefore improve the timing analysis. For systems where tasks are allowed to migrate between cores, cache related migration delay (CRMD) has been studied in [Hardy and Puaut 2009]. This work estimates the number of cache lines that can be reused from the L2 shared cache when a task migrates from one core to another. Due to the lack of analysis techniques for multicore platforms with a complex hierarchy of shared caches, an empirical study has been proposed in [Bastoni et al. 2010] to evaluate the impact of cache-related preemption and migration delays (CPMD).

In contrast to timing analysis techniques where caches are used without restrictions, the approach of managed caches have the advantage to avoid complex analysis methods for estimating the cache behavior. Indeed, the time predictable architecture in [Paolieri et al. 2009] proposes a statically-partitioned L2 cache to avoid the inter-core cache conflicts. In addition, manged caches can be used in situations where the static analysis cannot be used, for example, the case where the cache replacement policy is not documented. On the other hand, while managing the cache space provides a timing isolation between tasks, the reduced cache space may impact the task execution time. We are not a ware of any work that compares the managed shared caches with statically analyzed caches.

2.6. Scratchpad Memories

Finally, it must be noted that *scratchpad* memories have been proposed as a more predictable alternative to caches, especially but not limited to real-time systems. The scratchpad memory is a special static RAM memory placed close to the processor (on-chip, similar to L1-cache). The address space of the scratchpad is mapped onto predefined memory address of the processor. Unlike caches, the scratchpads have to be explicitly managed. In other words, the memory blocks have to be moved in software from main memory and copied into the scratchpad before being used. Thus, scratchpads are highly predictable in the sense that they have one access latency compared to caches with two different latencies for cache hit and miss. However, the need for explicit program management means that legacy programs cannot be easily executed on scratchpad-based systems; for this reason, support for scratchpad memory in available commercial systems is limited. Since the focus of this survey is specifically on management mechanisms for cache-based systems, we do not cover scratchpad-based systems in details in the following sections; however, we provide a brief summary of work concerning scratchpad allocation below to allow a comparison with cache systems.

In general, cache locking and scratchpad allocation have similar objectives: to control the set of memory blocks present in local memory at any time. However, as noted in [Puaut and Pais 2007; Whitham and Audsley 2009], there are significant differences

between caches and scratchpad. First of all, scratchpads do not suffer from intra-task interference, since the allocation of memory blocks is entirely under the control of the programmer/compiler; cache locking cannot prevent conflicts due to too many cache lines being allocated to the same associative set. However, since scratchpads are not transparent with respect to address translation, management schemes have to impose significant constraints on analyzable code; in particular, memory aliases must be statically resolved, since otherwise the management scheme risks loading the same data into two different positions in the scratchpad. A second important difference is related to the type of fragmentation. Cache-based systems suffer from internal fragmentation, since they cannot load data blocks smaller than the size of a cache line. Scratchpad-based systems suffer from external fragmentation, since they are forced to load memory blocks of varying size into contiguous memory locations in the scratchpad. As shown in [Puaut and Pais 2007], the relative performance of locked caches vs scratchpads is thus dependent on the size of the considered memory blocks. Due to the discussed differences, we argue that most techniques and results for cache-based systems cannot be directly applied to scratchpad-based systems and vice-versa.

3. INDEX-BASED CACHE PARTITIONING METHODS

There are two index-based cache partitioning categories: hardware- [Kirk and Strosnider 1990; Liu et al. 2004; Iyer 2004; Rafique et al. 2006; Suhendra and Mitra 2008; Srikantaiah et al. 2008] and software-based [Wolfe 1994; Liedtke et al. 1997; Chousein and Mahapatra 2005; Guan et al. 2009; Lu et al. 2009; Muralidhara et al. 2010; Kim et al. 2013] techniques. The former requires special hardware support, such as specialized implementations, that are not available in most of the current commercial processors. The latter has the advantage of being fully transparent to applications and there is no need for special hardware support. Software-based partitioning is further divided in those that require OS or compiler support.

3.1. Hardware index-based partitioning.

Several hardware index-based cache partitioning have been proposed to improve the average-case performance of applications. However, some of these approaches could also be applied to SRT. Below we discuss the main hardware index-based partitioning approaches and classify each approach in either average-case or HRT.

Average-case performance. The Shared Processor-Based Split L2 cache organization assigns cache sets according to the CPU ID [Liu et al. 2004]. The L2 cache controller (configurable by the OS) keeps a table that maps the CPU ID to its sets. Every CPU memory access looks first at its available sets, but can subsequently look at other CPU sets before going to the main memory. Upon a miss, the requested memory block is allocated only into the available CPU sets. Set pinning is a similar approach, where cache sets are associated with owner cores [Srikantaiah et al. 2008]. However, in set pinning, memory accesses that would lead to inter-core interference are redirected to a small core owned private (POP) cache. Each core has its POP cache, storing memory blocks that would cause inter-core cache misses. The objective is to reduce these cache misses and improve the average system performance.

CQoS classifies the applications memory accesses in priorities and then assigns more set partitions to higher priority applications [Iyer 2004]. The CQoS framework also implements a selective cache allocation in which it counts the number of lines occupied in the cache at a given priority level and probabilistically allocates or rejects cache line allocation requests.

Rafique *et al.* proposed a hardware implementation of a quota enforcement mechanism [Rafique et al. 2006]. The quotas is enforced at a set-level for different tasks/applications that access the shared cache. The mechanism requires the maintenance of

an ownership information for each cache block, which is performed by adding a Tag-owner-ID along with each tag in the cache address bits.

Hard real-time systems. Strategic Memory Allocation for Real-Time (SMART) was the first hardware-based implementation of a cache mechanism designed to provide predictability for uniprocessor real-time systems [Kirk and Strosnider 1990]. SMART divides the cache into M segments. Then, these M segments are mapped into N tasks. Shared data structures that require coherent cache accesses are placed in a specific partition, named shared partition. This shared partition can be formed by a set of segments [Kirk and Strosnider 1990]. Performance critical tasks have private partitions. Private partitions are protected and all tasks can access the shared partition. In order to identify the number of segments assigned to a task, a cache location address has also a cache ID field. Moreover, a hardware flag identifies whether a task accesses a private or a shared partition. The cache ID, segment count field, and hardware flags are part of each task's context and are loaded on every context switch performed by the OS. The authors presented a SMART designed for the MIPS R3000 processor.

Chousein and Mahpatra proposed a hardware-based cache partitioning for fully-associative cache architectures [Chousein and Mahapatra 2005]. The cache partitioning mechanism should provide a mean to search and isolate a cache line efficiently. Thus, a partition segment is created by aggregating multiple memory entries and associating them with a single tag entry [Chousein and Mahapatra 2005]. Content addressable memory (CAM) cells together with ternary content addressable memory (TCAM) cells form each tag entry. The use of CAM and TCAM reduces miss ratio.

Suhendra and Mitra proposed a cache partitioning mechanism able to perform three different partitioning strategies: "(1) no partition, where a cache block may be occupied by any task, scheduled on any core; (2) task-based partitioning, where each task is assigned a portion of the cache; or (3) core-based partition, where each core is assigned a portion of the cache, and each task scheduled on that core may occupy the whole portion while it is executing" [Suhendra and Mitra 2008]. Cache partitioning is then combined with a static cache locking mechanism (a cache line cannot be removed from the cache) or a dynamic scheme (a cache line can be reloaded at run-time). The experimental evaluation performed in a dual-core processor (simulated) with 2-way set-associative L1 and L2 caches resulted in a set of guiding design principles for real-time systems: (1) the best cache partitioning method that should be used with a static cache locking is core-based partition; (2) core-based partitioning is the best option independent of locking strategy; and (3) when comparing dynamic and static cache locking, the dynamic approach is better only when tasks have a considerable number of hot regions and when the size of the shared cache is not big.

3.2. Software index-based partitioning

The most common software-based cache partitioning technique is page coloring [Liedtke et al. 1997; Tam et al. 2007; Guan et al. 2009]. Page coloring explores the virtual to physical page address translations presented in virtual memory systems at OS-level, when caches are physically-indexed. Page addresses are mapped to predefined cache regions, avoiding the overlap of cache spaces. Figure 3 illustrates the physical addresses from the cache and OS point-of-views for the PL-310 (a.k.a. L2C-310) L2 cache controller adopted in ARM Cortex-A9 MPCore chips. By controlling the colored bits of the set-associative cache number, the OS can change the mapping of 4 KB pages in the physical memory and the cache location. The PL-310 cache controller can be synthesized with a 1 MB shared 16-way set-associative L2-cache with 32-bytes per line. There are 2^{11} sets in the cache (1 MB/16ways $\times$ 1way/32 B). Thus, the first 5 bits in the cache address access a byte in a cache line, the next 11 bits access a set, and the remaining 16 bits define a line from one of the 16 ways (Tag in Figure 3).

The idea of page coloring is to assign color 0 to page 0, color 1 to page 1, and so on, starting again from color 0 after reaching the maximum color number. In the same PL-310 chip, there are 128 cache lines in each 4 KB page. Each of these lines has a different cache set index. Since 4 bits are available to page coloring (cache size / number of ways / page size), page 16 maps to the same color as page 0. It is possible to partition the cache by assigning different colors to tasks/cores. Below, we classify works that use software index-based partitioning in those proposed for improving the average-case performance and those proposed for proving real-time guarantees. In the latter case, we also classify the work regarding its implementation (OS or compiler-based) and identify whether it targets SRT or HRT system.

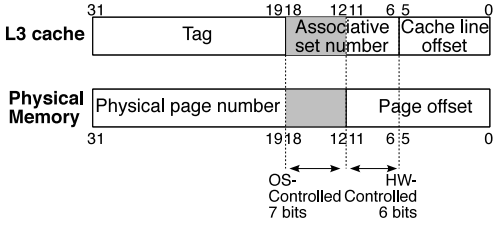

Fig. 3. Physical address view from the cache (on top) and from the OS (bottom)

Average-case performance. The first implementation of a page coloring mechanism targeting performance was carried out using the MIPS OS [Taylor et al. 1990; Zhang et al. 2009]. Then, page coloring was used to reduce cache misses in single applications by assigning different colors to applications and consequently, distributing the memory accesses through the entire cache [Kessler and Hill 1992; Romer et al. 1994; Sherwood et al. 1999]. More recently, several shared cache partitioning mechanisms based on page coloring for general-purpose systems have been proposed [Tam et al. 2007; Lin et al. 2008; Zhang et al. 2009; Guan et al. 2009].

Tam *et al.* used information collected by the hardware performance counters to estimate the size of each shared L2 cache partition [Tam et al. 2007]. Experimental results have shown that cache partitioning can recover up to 70% of degraded instruction per cycles due to cache contention. Lin *et al.* implemented a page coloring mechanism in the Linux kernel for x86 processors [Lin et al. 2008]. The authors changed the Linux memory management to support multiple lists. Each list is filled up with pages that have the same color. To search for a page when a process gets a page fault, the kernel uses a round-robin method to scan the lists with allocated colors. Two cache partitioning policies were supported: static and dynamic. The static policy defines the memory colors assigned to each process at the beginning of its execution. In the dynamic policy, a page recoloring is enforced whenever a process desires to increase its cache resource. Page recoloring is carried out by rebuilding the virtual to physical memory mapping. Page recoloring involves data copy from an old page to a new page, which has a new color. Although page recoloring achieved good QoS for general-purpose applications, the process of copying pages in a real-time application may result in deadline misses. Zhang *et al.* proposed a hot-page coloring approach [Zhang et al. 2009]. Hot pages are identified by checking the page table entry access bit. Whenever a page is accessed, its access bit is automatically changed by the MMU. The estimation of the frequency of access to a page is done by checking and clearing the access bit periodically. This approach also supports page recoloring for dynamic execution environments.

Bugnion *et al.* implemented page coloring support in the SUIF parallelizing compiler [Bugnion et al. 1996]. The objective was to improve performance of general-

purpose multicore systems. The proposed technique, named compiler-directed page coloring (CDPC), predicts access patterns of a compiler-parallelized application, such as array access patterns. Information from the compiler is used to perform cache partitioning using a page coloring mechanism, tailored by each application. Basically, the OS generates preferred color for each virtual page. Additionally to the compiler information, the OS also receives machine-specific parameters, such as the number of cores, cache configuration, and page size. Then, the OS tries to honor all received information as much as possible. Although this work has not been proposed to directly deal with real-time systems, it could be adapted to be used in SRT systems.

Real-time approaches. Several works proposed software-based cache partitioning to increase predictability of real-time systems [Wolfe 1994; Mueller 1995; Liedtke et al. 1997; Guan et al. 2009; Kim et al. 2013; Ward et al. 2013]. We classify these works in two categories: those that are implemented at the OS-level and those that are implemented at the compiler-level. We discuss both categories below.

Hard real-time OS index-based partitioning. OS-controlled cache partitioning for real-time systems was first proposed by Wolfe [Wolfe 1994]. Wolfe proposed a mechanism similar to page coloring to provide predictable execution times for low-priority tasks in a preemptive uniprocessor system [Mueller 1995]. In fact, his approach alters the address decomposition into tag, index, and offset as in the page coloring partitioning [Mueller 1995]. The technique requires that the task set is static and that all programs are compiled by the same compiler prior to the system start. Liedtke *et al.* used page coloring to provide predictability for uniprocessor real-time systems [Liedtke et al. 1997]. Bui *et al.* considered the cache partitioning problem "as an optimization problem whose objective is to minimize the worst-case system utilization under the constraint that the sum of all cache partitions cannot exceed the total cache size" [Bui et al. 2008]. The authors proposed a genetic algorithm to solve the optimization problem. The cache partitioning mechanism is based on page coloring. Experimental evaluation has shown an improvement on the schedulability of single-core systems.

Guan *et al.* proposed a cache-aware multicore real-time scheduling algorithm, named FP_{CA}, that divides the shared cache space into partitions [Guan et al. 2009]. The used cache partitioning mechanism is page coloring. Tasks are scheduled in a way that at any time, any two running tasks' cache spaces (*e.g.,* a set of partitions) are non-overlapped. A task can execute only if it gets an idle core and enough cache partitions. The authors proposed two schedulability tests, one based on a linear problem (LP) and another one as an over-approximation of the LP test. Tasks are not preemptive and the algorithm is blocking, i.e., it does not schedule lower priority ready jobs to execute in advance of higher priority even though there are enough available resources.

Kim *et al.* stated that cache partitioning based on page coloring suffers from two problems (1) the memory co-partitioning and (2) the limited number of partitions [Kim et al. 2013]. Then, the authors proposed a cache management scheme in which they assign to each core a set of private partitions to avoid inter-core cache space interference. However, tasks within each core can share cache partitions. Although this can result in intra-core interference, it solves the aforementioned problems. In other words, each task can now have a larger number of partitions which could improve its execution performance. In addition, memory partitions can be utilized by all tasks. They bound the penalties due to the sharing of cache partitions by accounting for them as cache-related preemption delays when performing the schedulability analysis.

Hard/Soft real-time OS index-based partitioning. MC^2 treats memory colors as shared resources to which accesses must be arbitrated by either a real-time locking protocol or a scheduling algorithm [Ward et al. 2013]. The OS associates a set of colors to each task. With locking mechanism, a job must acquire a lock for each color it needs before execution, and it releases these locks when it finishes execution. Thus, the job

entire execution is treated as a critical section, which may create a long priority inversion blocking. To mitigate this problem, the authors propose period splitting and job splitting as two ways to reduce the detrimental effect of lengthy critical sections. However, this may cause jobs to reload their working sets.

Unlike the locking mechanism described above, the cache colors can be treated as preemptive resources in which the concurrent accesses can be mediated by scheduling. Instead of dealing with a scheduling problem over two preemptive resources: the processor and the cache, the authors reduce the problem into uniprocessor scheduling. They define a logical *cache processor* to whom they assign tasks that share either cache colors or processor. They then evaluated the schedulability of each cache processor as a uniprocessor, and apply known schedulability tests. The authors compared both techniques in terms of schedulability of the Partitioned Rate-Monotonic (P-RM) algorithm and concluded that cache locking approach is better than the scheduling approach. The target real-time system is composed of HRT and SRT tasks.

Gracioli and Fröhlich uses page coloring to create application and OS heaps composed of only pages with the same colors [Gracioli and Fröhlich 2013]. User-level memory requests are served by application heaps, while OS-level memory requests are served by the OS heap. Thus, the work evaluated the cache interference caused by the RTOS into the schedulability of real-time tasks (SRT or HRT). For a lightweight RTOS, cache partitioning reduces cache interference in a level that applications are not delayed. The authors also proposed a user-level memory allocation mechanism.

Hard real-time compiler index-based partitioning. Mueller was the first to introduce compiler support for cache partitioning in the context of real-time systems [Mueller 1995]. The compiler receives as additional inputs the cache size and the partition size of a task. The final object files are separated in code and data partitions for each task, and are combined into an executable by the linker. To deal with code partitions larger than the partition size, the compiler inserts at the end of each code partition an unconditional jump to the next code partition. According to Mueller [Mueller 1995]: “each partition is stored in a separate object file, which may be padded with no-ops at the end to extend it to the exact size given by the cache partition size. Global data is split into memory partitions of the data cache partition size. The compiler ensures that no data structure spans multiple partitions. If the size of a data structure exceeds the cache partition size, it is split over multiple partitions and the compiler needs to transform the access to the data structure”. Local data on the stack is split into partitions by manipulating the stack pointer whenever necessary [Mueller 1995]. Dynamic allocation on the heap is supported as long as memory requests do not exceed the cache partition size [Mueller 1995]. OS and libraries are also treated as separated partitions. The used cache partition mechanism is based on page coloring and the approach targets uniprocessor real-time systems.

Vera *et al.* used compiler techniques together with cache partitioning and locking to improve the predictability in preemptive multitasking uniprocessor systems in the presence of data caches [Vera et al. 2003b]. The proposed predictable framework works both with hardware or software cache partitioning schemes. The compiler technique uses loop tiling and padding to reduce capacity and conflict misses, that is, it reduces intra-task interference. Loop tiling reorders accesses, which shortens the reuse distance, while padding modifies the data layout of arrays and data structures. Cache locking is used to lock cache lines that are accessed by more than one task. The authors compared the proposed framework with static cache locking in which all tasks share the whole cache in terms of worst-case performance of a multitasking system. The results indicated that framework schedules tasks that need a high throughput.

Cache Replacement Policy Analysis. In general, most discussed index-based partitioning methods are mainly concerned with avoiding intra-core and/or inter-core in-

terference, by assigning disjoint index sets to either different tasks on a given core or to different cores. However, if locking is not used, a task can still suffer from intra-task interference: loading a memory block of the task in an assigned associative set could cause another memory block in the same associative set to be evicted.

To bound the number of such self-evictions that a task can suffer, a significant amount of effort has been spent to analyze the effects of various cache replacement policies in the context of static WCET analysis for HRT systems (for example, see [Ferdinand 1997] for LRU, [Grund and Reineke 2010a; Guan et al. 2013] for FIFO, and [Grund and Reineke 2010b] for PLRU). Based on the seminal work in [Ferdinand 1997], such techniques typically try to classify memory accesses in cache as either hits or misses by constructing *must* and *may* sets of memory blocks: at any given point in time, the *must* set represents the set of memory blocks that are known to be in cache, while the *may* set represents the set of blocks that may be in cache. Then, accesses to blocks in the *must* set can be safely categorized as hits, accesses in the complementary of the *may* set are misses, and accesses that are in neither are unknown.

A related problem is to determine the maximum number of memory blocks that can be utilized in cache without conflicts [Pellizzoni 2010]. For instance, a task can not safely allocate more than one memory block per set with a random replacement policy, since otherwise in the worst-case, fetching one block of the task could always evict another block in the same associative set. On the other hand, for a n-way associative cache both LRU and FIFO allow a task to safely allocate n blocks per set, that is, the task can utilize the entire cache size.

3.3. Summary

Table I summarizes the discussed index-based cache partitioning works that rely on hardware-specific implementations. The works proposed by [Liu et al. 2004], [Srikantaiah et al. 2008], [Iyer 2004], and [Rafique et al. 2006] although focused on multicore systems, were not proposed for real-time systems and, consequently, predictability is not the main concern. Instead, they aim at improving the QoS and the average execution time. However, they could be applied to SRT systems as well. SMART was the first hardware-based implementation of a cache mechanism designed to provide predictability for uniprocessors [Kirk and Strosnider 1990]. The SMART approach consists of dividing the cache into M segments and assigning these segments individually to tasks. A cache location address is divided in a segment and cache ID fields that identify how many segments a task owns and which they are. Chousein and Mahpatra were the first authors to propose a hardware-based implementation of a cache partitioning mechanism to improve the predictability of multicore systems [Chousein and Mahapatra 2005]. Their approach focused on physical changes, using different technologies (CAM and TCAM cells) to implement cache memories. Suhendra and Mitra were the first authors to combine a cache partitioning mechanism, that can be implemented in hardware or software, with cache locking [Suhendra and Mitra 2008]. In their work, the cache partitioning mechanism performs task-based and core-based partitioning.

Table II summarizes the discussed index-based cache partitioning works that use a software approach, either implemented in the OS or in the compiler. The first works to use cache partitioning implemented in the OS did not focus on multicore real-time systems [Taylor et al. 1990; Kessler and Hill 1992; Romer et al. 1994; Sherwood et al. 1999]. Instead, their objectives were to improve the average performance in uniprocessors. Also, several works improved the average performance in multicore systems [Tam et al. 2007; Lin et al. 2008; Zhang et al. 2009]. Wolfe was the first author to use a cache partitioning mechanism to improve the predictability in uniprocessors [Wolfe 1994]. The work proposed by Liedtke et al. was the first to use page coloring, also in the context of uniprocessor real-time systems [Liedtke et al. 1997]. Bui et al. pro-

Table I. Comparative table of the reviewed state-of-the-art mechanisms of index-based cache partitioning with hardware-specific implementations.

Cache partitioning	Features				
	AVG, SRT, or HRT	**Inst. or data caches**	**Technique**	**Use cache locking**	**Multicore or singlecore**
[Liu et al. 2004]	AVG	both	cache controller table	no	multicore
[Srikantaiah et al. 2008]	AVG	both	POP cache	no	multicore
CQoS [Iyer 2004]	AVG	both	selective cache alloc.	no	multicore
[Rafique et al. 2006]	AVG	both	quota enforcement	no	multicore
SMART [Kirk and Strosnider 1990]	HRT	data	M segments	no	singlecore
[Chousein and Mahapatra 2005]	HRT	data	CAM and TCAM cells	no	multicore
[Suhendra and Mitra 2008]	HRT	data	HW or SW partitioning	yes	multicore

posed a genetic algorithm to optimize the worst-case system utilization by finding the best cache partitioning assignment for a task set [Bui et al. 2008]. Guan et al. proposed a cache-aware multicore scheduling algorithm that uses page coloring [Guan et al. 2009]. Kim et al. used page coloring to propose a cache management scheme that assigns to each core a set of partitions [Kim et al. 2013]. Kenna et al. proposed the cache management strategy called MC^2, which treats the management of cache lines as scheduling and synchronization problems, targeting a real-time system formed by HRT and SRT tasks [Ward et al. 2013]. All the discussed works so far used an OS implementation of a cache partitioning mechanism based on page coloring. Other works proposed changes in the compiler to support cache partitioning at the compile time. Mueller was the first author to introduce a compiler support for cache partitioning in the context of uniprocessor real-time systems [Mueller 1995]. Bugnion et al. implemented page coloring support in the SUIF parallelizing compiler targeting multicore processors [Bugnion et al. 1996]. Both Bugnion et al and Mueller's approaches used page coloring to partition the cache. Finally, Vera et al. used compiler techniques together with cache partitioning and locking to improve the predictability in preemptive multitasking uniprocessor systems in the presence of data caches [Vera et al. 2003b]. The mechanism relies on a hardware/software cache partitioning and was the first compiler-based mechanism to combine cache partitioning with cache locking.

4. WAY-BASED CACHE PARTITIONING METHODS

Way-based partitioning has two main advantages. First, the set-associative cache organization does not need to be changed, which does not have a dramatic impact on the overall structure. Second, this partitioning scheme isolates the requests for the different compartments from each other. Thus, no contention for the cache ways is suffered at the cores. However, the main limitations of way-based cache partitioning methods are the limited number of partition and granularity of allocations due to the associativity of the cache. Increasing the associativity of a cache is not always feasible or efficient since a higher-order associativity determines an increase in the cache access time and tag storage space. Below we classify works that use a way-based cache partitioning method according to their objectives of either improving average-case performance or providing real-time guarantees.

Table II. Comparative table of the reviewed state-of-the-art mechanisms of index-based cache partitioning implemented in software.

Cache partitioning	**Features**					
	AVG, SRT, or HRT	**Inst. or data caches**	**OS or compiler**	**Technique**	**Use cache lock.**	**Multicore or singlecore**
[Taylor et al. 1990; Kessler and Hill 1992; Romer et al. 1994; Sherwood et al. 1999]	AVG	data	OS	page coloring	no	singlecore
[Tam et al. 2007; Lin et al. 2008; Zhang et al. 2009]	AVG	data	OS	page coloring	no	multicore
[Bugnion et al. 1996]	AVG	both	compiler	CDPG	no	multicore
[Wolfe 1994]	HRT	data	OS	similar to page coloring	no	singlecore
[Liedtke et al. 1997]	HRT	data	OS	page coloring	no	singlecore
[Bui et al. 2008]	HRT	data	OS	genetic alg. with page col.	no	singlecore
[Guan et al. 2009]	HRT	data	OS	scheduling and page col.	no	multicore
[Kim et al. 2013]	HRT	data	OS	page coloring	no	multicore
[Ward et al. 2013]	SRT/HRT	data	OS	page coloring	no	multicore
[Mueller 1995]	HRT	both	compiler	compiler with page col.	no	singlecore
[Vera et al. 2003b]	HRT	data	compiler	HW or SW cache partitioning	yes	singlecore

4.1. Average-case performance

Ranganathan *et al.* proposed a cache architecture that allows dynamic reconfiguration of partitions [Ranganathan et al. 2000]. Specifically, the traditional structure of a set-associative cache is adapted to allow dynamic definition of partitions which can be assigned to given address ranges. The dynamic reconfiguration can be performed at runtime by software. In the proposed architecture, each cache partition can be the result of the aggregation of one or more cache ways, leading to a way-based partitioning scheme. Additional cache logic is required to differentiate between address spaces/partitions; to multiplex the tag comparators; and to generate different miss/hit signals for each addressed partition. However, the additional logic and wiring has been proven to minimally affect the cache access time, with a variable overhead that is proportional to the ratio $\frac{\#\ of\ partitions}{cache\ size}$. Although the described scheme has not been designed for real-time applications, a straightforward extension could be designed for multi-core systems, allowing to bind the address space of a task running on a given core to a given cache partition. This would eliminate the inter-core interference for all those data that reside in an exclusively owned partition. However, intra-task interference (self-evictions) could still be possible.

Chen *et al.* proposed a technique to partition each cache set into shared and private ways [Chen et al. 2009]. The technique classifies all cache lines in shared or private. Let A be the shared cache associativity and Q be a quota for shared cache lines, partitioning is performed in such a way that the shared cache lines occupy Q way in each set while private lines occupy A-Q [Chen et al. 2009]. By assigning different partitions to shared and private data, they reduced the LLC miss rate and increased QoS.

Muralidhara *et al.* proposed a dynamic software-based partitioning technique that partitions the shared cache among threads of an application [Muralidhara et al. 2010]. At the end of each 15 ms interval, the dynamic cache partitioning scheme uses hardware performance counters information, such as the number of cache hits and misses, the number of cycles, and retired instructions for each thread, to allocate different cache spaces based on individual thread performance. The objective is to speed up the critical path, that is, the thread that has the slowest performance and, consequently, improve the overall performance of the application. To perform cache partitioning, the authors use either a reconfigurable cache, which modifies the cache hardware structure during the execution, or modify the cache replacement algorithm, which partitions the cache gradually. According to Muralidhara *et al.*: "when a thread suffers a cache miss and the number of cache ways that belong to it is less than the thread's assigned cache partition ways, a cache line belonging to some other thread is chosen for replacement. If the number of cache ways belonging to the thread is greater than or equal to the assigned number of ways, a cache line belonging to the same thread is chosen for replacement" [Muralidhara et al. 2010]. Consequently, the replacement policy gradually partitions the shared cache. Experimental results have shown a performance gain of up to 23% over a statically partitioned cache [Muralidhara et al. 2010]. The work does not provide real-time guarantees.

Similarly to [Chen et al. 2009], in [Sundararajan et al. 2013], Sundararajan *et al.* propose RECAP: an architecture for a set-associative cache that can be partitioned across cores at the granularity of a single cache way. Different cache ways are assigned to shared and private data, based on the observation that the majority of memory accesses for parallel applications is performed in shared memory, while above 90% of cache data belong to private memory regions. Moreover, RECAP performs an automatic arrangement of the partitioned ways, so that private data assigned to specific cores are allocated on ways that are positioned on the left side of the cache, while shared data are allocated on ways starting from the right end of the cache. The enforced data-alignment allows to keep the unused ways in the center of the cache and thus allowing dynamic power-saving by powering down unused ways. Partition arrangement for each core is performed in order to maximize utilization of the cache ways: a heuristic algorithm monitors the number of misses for each core and determines how many cache blocks need to be assigned in order to capture a given fraction of cache misses. Finally, the computed ways-to-cores assignment is enforced on the proposed cache architecture by programming a series of configuration registers. The results show a 15% average performance increase for large applications, with a reduction of power consumption which is above 80%. However, even though this approach is effective to limit inter-core interference, its scope is beyond predictability, making it suitable for soft real-time rather than safety-critical applications. First because intra-core/intra-task interference is not addressed; second due to the heuristic approach that is being used to perform ways-to-cores assignment at run time.

Varadarajan *et al.* in [Varadarajan et al. 2006] propose the idea of "molecular caches": CPU caches that are organized as a series of molecules. In this architecture, molecules are small in size (8-32KB) and reflect the structure of a direct-mapped cache. Molecules are grouped into tiles that can be assigned statically or dynamically to cores. Thus, by definition, per-core partitions are defined at the granularity of tiles, that in

turn represent an aggregation of cache ways. Moreover, in order to address intra-core interference, individual regions can be defined internally to each tile and assigned to a specific application. In order to do this, each molecule of the same region inside a given partition exposes a configuration register which can be programmed with a unique identifier of a running application. In this way, all the molecules of a given region will be exclusively used by the matching application. The proposed scheme results effective in isolating cache misses from different running applications on the same or different cores, but the complexity of the resulting structure may not scale well with cache size.

Since molecules (and thus regions) can be dynamically assigned to applications (or unassigned), a high degree of runtime reconfiguration is possible. As such, assignment decision need to be made by a software module that runs periodically or in an adaptive manner. The metric used in the assignment controller is keeping the experienced number of caches misses as close as possible to a threshold that is statically defined for each application. Even though inter-task interference is not addressed by such a partitioning scheme, molecular cache represent an effective solution to prevent inter/intra-core interference, while providing the cache with a fine-grained partitioning mechanism. Moreover, the particular choice of associativity and size for the molecules leads to a circuit design that enables a reduction in energy dissipation. However, due to its complexity, a hardware implementation of molecular caches is not available to date. The applicability of such an architecture to real-time applications would be broad, assuming that the assignment controller employs a predictable allocation strategy.

In [Qureshi and Patt 2006], a utility-based approach is proposed to dynamically partition the cache on a demand basis. The main insight is that differences in the working set size as well as memory addressing patter of applications affect the way they benefit from cache assignment. Specifically, applications that exhibit low spatial and temporal locality in their memory access pattern, combined with a large memory footprint size, benefit less from cache resources than applications with opposite characteristics. Thus, the former can be classified as low-utility applications, while the latter can be considered as high-utility applications from the cache assignment point of view. To account for cache utility, Quereshi and Patt [Qureshi and Patt 2006] propose adding a low-overhead circuit in the cache controller. The addition consists of a number of utility monitors (one per each core accessing the shared cache) and a single circuit that runs the partitioning algorithm relying on data collected at the utility monitors. The partitioning algorithm reads all the hit counters from the different utility monitors and computes a cache partitioning at the granularity of a way that minimizes the overall number of misses suffered by all the applications.

Evaluation results show that such partitioning approach is able to provide fairness and performance speedup if compared to using LRU and static partitioning (*e.g.,* an even split of the cache in two halves for two cores). The hardware overhead is relatively low when applied to LRU caches, but can significantly increase for different cache replacement policies. However, the inherently heuristic approach employed in the partitioning algorithm is not directly suitable for HRT purposes, since the amount of assigned cache strictly depends on the workload on other cores. This problem could be mitigated allowing a minimal assignment of cache per each core or application.

In [Suh et al. 2004] a dynamic cache partitioning scheme is presented. In this scheme, cache space is explicitely allocated amongst simultaneously executing processes with the goal of minimizing the overall cache misses. In order to attain this goal, the cache controller is augmented with a set of online counters. Next, each process' gain or loss under different cache allocations is considered. By using the described mechanism, the hardware dynamically adjusts the allocation to match process needs. Allocation is performed at the granularity of single cache ways. This technique mitigates the effects of cache pollution due to inter-core interference in multicore systems.

However, the inherently heuristic nature of this work makes it not directly suitable for real-time systems. In fact, the amount of cache dynamically allocated to a given task depends on the behavior of all the other active tasks in the system.

4.2. Real-time approaches

The only work originally proposed to improve the predictability of both SRT and HRT systems that uses a way-based cache partitioning was proposed by Chiou *et al.* [Chiou et al. 2000]. The authors introduced column caching, a mechanism that allows software to specify that certain data is restricted to be placed into specific way defined through a bit-vector. Column caching requires a hardware modification to the replacement unit of the cache in order to limit replacement to the column (way) specified by the bit-vector stored in page table entries. Thus, the translated address will include the set number as well as the way number. If combined with page-coloring as in index-based methods, the whole cache (both sets and ways) can be controlled and utilized.

4.3. Summary

All the reviewed way-based cache partitioning mechanisms use a hardware-specific implementation. They are summarized in Table III. Most of the way-based cache partitioning works were not originally proposed to improve the predictability of real-time systems, although some of the proposed techniques could be adapted in the context of SRT systems with minor modifications to the proposed techniques considering worst-case task behavior.

Table III. Comparative table of the reviewed state-of-the-art way-based cache partitioning mechanisms.

Cache partitioning	**Features**						
	AVG, SRT, or HRT	**Inst. or data caches**	**HW-only**	**HW-SW**	**Technique**	**Use cache lock.**	**Multicore or singlecore**
[Ranganathan et al. 2000]	AVG	both	no	yes	dynamic reconfiguration	no	singlecore
[Chiou et al. 2000]	SRT/HRT	both	no	yes	changing the replacement policy	no	singlecore
[Chen et al. 2009]	AVG	data	yes	no	quota enforcement	no	multicore
[Muralidhara et al. 2010]	AVG	data	no	yes	dynamic cache part.	no	multicore
RECAP [Sundararajan et al. 2013]	AVG	data	yes	no	specific mem. arch.	no	multicore
[Varadarajan et al. 2006]	AVG	data	no	yes	molecular cache	no	multicore
[Qureshi and Patt 2006]	AVG	data	yes	no	utility-based approach	no	multicore

5. CACHE LOCKING METHODS

Cache locking prevents the eviction of cache lines by marking them as locked until an unlock operation is performed. Cache locking is a hardware-specific feature that it is not present in all of the modern multicore processors. There are two ways to lock a cache content: (A) through an atomic instruction to fetch and lock a given cache

line into the cache, or (B) defining, for each single CPU in the system, the lock status of every cache way. The last mechanism is called *lockdown by master* in multicore systems [Mancuso et al. 2013]. Some embedded multicore platforms feature only an atomic instruction, such as Freescale P4040 and P4080 platforms, while other platforms, such as TI OMAP4460 and OMAP4430, Nvidia Tegra 2 and 3, Xilinx Zynq-7000, and Samsung Exynos 4412, implement a lockdown by master mechanism.

To exemplify the utility of a cache locking mechanism, consider a dual-core platform with a 2-way set-associative cache and a cache controller that implements a lockdown by master mechanism. We could 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 a task running on CPU 1 would deterministically allocate blocks on way 1 and blocks allocated on way 1 could never be evicted by a task running on CPU 2. The same situation occurs on way 2 referring to CPU 2. This assignment can be easily changed at run-time 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 [Mancuso et al. 2013]. Hence, cache locking provides a more predictable and controllable access to shared caches, easing the WCET estimation and improving the performance of real-time applications.

We classify the works that use cache locking as hardware-only, when a specific hardware design is proposed, or hardware-software approaches, when the hardware available on current multicore processors is used by a software layer (RTOS, compiler, or specific algorithms for instance). We further state if the work provides a cache locking only for instruction caches, data caches, or both caches and if the work targets only SRT systems, HRT systems, or both.

5.1. Hardware-only approaches

Asaduzzaman *et al.* proposed a miss table (MT) at the L2-cache level using cache locking [Asaduzzaman et al. 2010]. The cache memory addresses that cause the most number of misses if not locked are kept in the MT. The cache miss for each block address is obtained by the Heptane simulation tool [Avila and Puaut 2009]. Then, the MT sorts the block addresses in descending order of the cache miss numbers. The proposed MT-based scheme was evaluated through the simulation of an 8-core processor with 2 levels of cache using the MPEG4 and H.264 decoding and FFT algorithms. The results have shown a predictability improvement in all algorithms. This technique can be applied to either SRT or HRT systems.

Sarkar *et al.* proposed a predictable task migration scheme using cache locking for the PFair scheduling algorithm, targeting HRT systems [Sarkar et al. 2011]. The authors proposed several cache migration models in the presence of a cache locking mechanism to deterministically bound the migration delay of tasks. The work uses the push model hardware feature, where every cache controller has a push logic block and each cache line has an identifier. When a task migrates, the new core receives a push request to start loading the cache lines for that task. Then, the modified hardware identifies locked lines pertaining to the migrated task and uses the cache line identifier to correctly manage locked cache lines. Simulation results have shown a reduction of the migration cost of up to 56%.

5.2. Hardware-software approaches

Campoy *et al.* were the first authors to use cache locking in the context of uniprocessor HRT systems [Campoy et al. 2001]. The authors proposed a genetic algorithm that selects which instruction blocks are loaded and locked in the cache in a preemptive, multitasking system. In such a system, the locking of a cache line from one task af-

fects the other tasks. The genetic algorithm estimates the WCET of each task with data loaded and locked in the cache. The algorithm uses the estimated WCET to determine the response time for each task. Experimental results indicated that the proposed algorithm calculates the response time of tasks with negligible overestimation and without performance loss, considering a preemptive scheduler [Campoy et al. 2001].

Puaut and Decotigny addressed intra-task and intra-core interferences using static cache locking of instruction caches in uniprocessor multitasking HRT systems [Puaut and Decotigny 2002]. The authors proposed two algorithm to select the instruction cache blocks that should be locked. In contrast to [Campoy et al. 2001], the two algorithms use memory access patterns of the instruction flow to determine which cache lines must be locked. The first algorithm aims at minimizing the CPU utilization of the task set, while the second aims at reducing the intra-core interference using a fixed-priority scheduler. The authors compared the two algorithms with a static cache analysis for large instruction caches and caches with a large degree of associativity. Both algorithms have presented a better performance [Puaut and Decotigny 2002].

A dynamic locking strategy for instruction caches was proposed in [Arnaud and Puaut 2006]. A control-flow graph is extracted from task binary. Next, basic blocks are identified and regions of code that aggregate one or more basic blocks are created. The authors propose an heuristic algorithm (Region Merging and Inlining) to perform close-optimal aggregations of basic blocks to regions. Each region has a locked/unlocked state that is set appropriately by the allocation algorithm considering the following parameters: a) mapping of region to cache lines; b) execution frequency; and c) performance gain derived from locking the considered region with respect to its loading/locking cost. The approach has been tested on an emulated MIPS R3000 platform.

Vera *et al.* combined compile-time cache analysis with data cache locking to improve the WCET estimation of uniprocessor HRT systems [Vera et al. 2003a]. An algorithm executed at compile-time identifies code regions where a static analysis does not precisely predict their behavior. The algorithm uses a reuse vector to perform cache locality analysis and to select data to be loaded and locked in the cache. Static analysis is used in the other code regions. Also, the compile-time algorithm inserts beyond the lock instructions, unlock and load instructions and compute the WCET for k-way set-associative data caches. The authors implemented the algorithm in the SUIF2 compiler and obtained a more predictable cache behavior with minimal performance loss.

Falk *et al.* proposed a technique that also explore static locking of instruction caches at compile-time to minimize the WCET in uniprocessor SRT/HRT systems [Falk et al. 2007]. Differently from the previous works, the technique uses the worst-case execution path in order to apply cache locking. An optimization algorithm receives a compiled and linked binary executable file as input. The target processor is ARM920T. Then, the algorithm calls a static WCET analysis tool (aiT), which extracts data that impact the WCET from the binary's function. From this data, the optimization algorithm selects the most relevant code (*i.e., functions*) that will be locked in the instruction cache. Experimental results reported a reduction between 54% and 73% in the WCET of some benchmarks running on the ARM920T processor.

Exploiting information extracted at compile-time, Liu *et al.* propose and analyze three algorithms to perform allocation of instruction cache area through locking [Liu et al. 2009a]. The proposed algorithms consider both static locking and dynamic locking, with the goal of minimizing worst-case CPU utilization through cache allocation. The results are for a set of tasks in a multi-task scenario, improving on what achieved for single-task systems in [Liu et al. 2009b]. Based on the achieved results, practical guidelines are provided to select the locking methodology that best fits specific workload characteristics. Experimental results show that close-optimal solutions for the

cache allocation problem can be found, improving the efficiency of the heuristic compared to what presented in [Falk et al. 2007].

Aparicio *et al.* proposed a dynamic cache locking strategy to load and lock the best cache lines into the instruction cache at each context switch [Aparicio et al. 2011]. The proposed approach deals with both intra-task and intra-core interferences. The method is based on an integer Linear Programming (ILP) method, is named Lock-MS (Maximize Schedulability), and targets uniprocessor HRT systems. The ILP-method analyzes a single task and the interference among tasks to load and lock the most relevant cache lines. At run-time, a dynamic cache locking method preloads the cache contents at every context switch, minimizing the worst cost of a preemption. The authors compared Lock-MU with a static locking mechanism [Puaut and Decotigny 2002] and obtained better performance results.

Ding *et al.* acknowledge the limitations of region-based dynamic locking and propose a more flexible locking strategy in [Ding et al. 2014]. Specifically, this work focuses on instruction caches and aims at performing a partial locking of the cache so to exploit the additional benefits of the unlocked cache lines. Moreover, instead of splitting the instruction blocks in regions that are (un)loaded in cache in a mutually exclusive manner, the paper introduces the notion of loop-driven locking. The key idea is the following: given a series of nested loops, each line selected for locking at an inner loop can also be locked/unlocked at the entry/exit point of any of the outer loops. The evaluation performed on the Chronos WCET tool reveal benefits with respect to: a) the static partial locking strategy proposed in a previous work by the same authors [Ding et al. 2012]; and b) region-based dynamic locking as proposed in [Arnaud and Puaut 2006].

Mancuso *et al.* proposes a memory framework that uses profiling techniques to analyze the memory access pattern of tasks and obtain the most frequently accessed memory pages [Mancuso et al. 2013]. Profiling is performed using memory observation tools and it is execution-independent. Then, page coloring is used to rearrange task pages into the physical memory address space so to optimize in-cache placement. Next, cache locking is used to provide isolation among tasks on the same core and on different cores, thus increasing predictability in HRT systems. The solution focuses on using existing hardware support on modern multi-core platforms and is implementable using locking by-line, by-master and by-way. The framework was implemented and evaluated in the Linux kernel. Evaluation was conducted using locking support in ARM Cortex-A9, and a porting to Freescale P4080 was performed in a successive extension of the work [Mancuso et al. 2015].

5.3. Summary

Table IV summarizes the discussed cache locking mechanisms. In [Asaduzzaman et al. 2010] and [Sarkar et al. 2011], two hardware-only cache locking mechanism are proposed. The first targets both multicore and single systems, while the second is proposed only for multicore systems. The rest of the works does not propose a hardware-specific mechanism to lock the cache. Instead, they use the cache locking mechanisms available on current processors. Campoy *et al.* were the first authors to study the effects of locking lines in instruction caches of uniprocessor real-time systems [Campoy et al. 2001]. The authors proposed a genetic algorithm to select the best lines to be locked. Following the same research line, in [Puaut 2002] the authors proposed two algorithms to also select best lines to be locked in an instruction cache. In [Vera et al. 2003a] and [Falk et al. 2007], the authors proposed changes in the compiler to extract information regarding the data/instruction access pattern by tasks. Then, this information is used to lock cache lines. Suhendra and Mitra (this work was discussed in Section 3) were the first authors to evaluate the combination of cache partitioning and cache locking in the context of multicore real-systems [Suhendra and Mitra 2008]. Specifically to cache

Table IV. Comparative table of cache locking state-of-the-art mechanisms.

Cache locking	Features						
	AVG, SRT, or HRT	**Instr. or data caches**	**HW-only**	**HW-SW**	**SW approach**	**Multicore or single-core**	**Static or dynamic**
[Asaduzzaman et al. 2010]	SRT/HRT	both	Yes	No	-	both	dynamic
[Sarkar et al. 2011]	HRT	both	Yes	No	-	Multicore	dynamic
[Campoy et al. 2001]	HRT	instr.	No	Yes	Genetic algorithm	Singlecore	static
[Puaut and Decotigny 2002]	HRT	instr.	No	Yes	Two alg. to select locked cache lines	Singlecore	static
[Vera et al. 2003a]	HRT	data	No	Yes	Compile-time alg.	Singlecore	static
[Arnaud and Puaut 2006]	HRT	instr.	No	Yes	Profiling, CFG analysis, heuristic	Singlecore	dynamic
[Falk et al. 2007]	SRT/HRT	instr.	No	Yes	Compile-time alg.	Singlecore	static
[Liu et al. 2009a]	HRT	instr.	No	Yes	Compile-time data, CFG analysis, heuristic	Singlecore	both
[Aparicio et al. 2011]	HRT	instr.	No	Yes	ILP method	Singlecore	dynamic
[Ding et al. 2014]	HRT	instr.	No	Yes	ILP-based WCET, loop-level block selection	Singlecore	dynamic
[Suhendra and Mitra 2008]	HRT	both	No	Yes	Cache partitioning and locking	Multicore	both
[Mancuso et al. 2013]	HRT	both	No	Yes	Profiling, cache partitioning and locking	Multicore	both

locking, the authors used the algorithms proposed in [Puaut 2002] to select the cache lines to be locked. Unlike this work, Mancuso *et al.* were the first authors to evaluate the combination of cache partitioning and locking using a real hardware and OS in the context of multicore real-time systems.

6. OPERATING SYSTEM MEMORY ALLOCATORS

Developers usually allocate data with little or any concern for cache memory hierarchy. Hence, the resulting data allocation in the cache memory hierarchy may interact poorly with the program's data access pattern [Chilimbi et al. 2000], occurring in intra-task, intra-core, and inter-core interferences. In a multicore real-time system, this bad memory allocation may cause the loss of deadlines and it is not tolerable. Thus, OS memory allocators for real-time systems must be predictable (despite the memory size being allocated) and cache-conscious in order to allocate memory efficiently. Basically, we can divide OS memory allocators for real-time systems in three categories: (i) those that are cache-aware, i.e., provide a cache partitioning or cache locking mechanism to ensure predictability; (ii) those that are predictable, i.e., the time to allocate memory blocks is bounded; (iii) those that are predictable and cache-aware. In this section we review OS memory allocators that are designed to be predictable and/or cache-aware.

6.1. Cache-aware allocators

Chilimbi *et al.* proposed a memory allocator (named *ccmaloc* – cache-conscious malloc) that receives the requested bytes and a pointer to an existing data object that the program is likely to access contemporaneously with the element to be allocated [Chilimbi et al. 2000]. The allocator attempts to allocate the requested data in the same cache

block as the existing item, thus improving data locality, cache hit rates, and the average execution time. The authors also integrated the memory allocator with a data structure reorganizer. The reorganizer basically transforms a pointer structure layout into a linear memory layout and maps structure elements to reduce cache conflicts using page coloring. The main drawback of this approach is the need for copying data to a new linear memory region and the absence of a predictable memory allocation time.

Cache-Index Friendly (CIF) also tries to improve the average execution time, instead of improving the predictability, by explicitly controlling the cache-index position of allocated memory blocks [Afek et al. 2011]. The central idea in CIF is to insert small spacer regions into the array of blocks within the allocator to better distribute block indices and disrupting the regular ordering of block addresses, returned by the allocator. The authors performed a set of experiments to show that CIF reduces intra-task and inter-core interferences. CIF, however, is not designed for real-time systems.

Gracioli and Fröhlich overloaded the C++ *new* operator to enable colored memory allocation/deallocation in an RTOS [Gracioli and Fröhlich 2013]. The overload of the *new* operator is part of the ISO C++ standard and is supported by any standard C++ compiler. The OS keeps a set of heaps, where each heap only contains pages with the same color. The user passes the color as parameter to the new operator, identifying from which heap the memory should be allocated (for instance, data = new (COLOR_0) (sizeof(int *)), allocates memory from the heap with pages colored as 0). The developer can also choose an OS-centric allocation scheme, in which the OS uses the thread ID to automatically choose from which heap data should be allocated (ID % maximum number of colors). The RTOS has its own colored heap, not causing interference with the application's colors. The approach does not have a predictable timing analysis.

The PALLOC allocator is primarily designed to assign disjoint sets of DRAM banks (private banks) to applications running on different cores [Yun et al. 2014]. This way, tasks running in parallel do not collide on DRAM banks and do not suffer inter-core conflicts at this level, as long as there is a sufficient number of banks to accommodate them. However, its implementation also allows enforcing a last-level cache partitioning based on the assignment of colored pages to tasks. PALLOC modifies the Linux buddy allocator so that specific page colors can be selected when allocating new memory pages. System designers can create multiple partitions and specify desired sets for each partition through the CGROUP interface. When a process in a CGROUP requires physical memory, PALLOC allocates only pages from the specified partition.

6.2. Time-predictable allocators

The Half-fit algorithm was the first dynamic memory allocator to perform allocation/deallocation in a constant and predictable time [Ogasawara 1995]. In Half-fit, free blocks of size in the range $[2^i, 2^{i+1})$ are grouped into a free list indexed by i. When a block is deallocated, it is immediately merged with neighboring free blocks. After merging, the index of the new free block of size r in a free list bit vector can be found using $i = \lfloor log_2\ r \rfloor$. This logarithm operation in a bit vector can be executed in one cycle in several 32-bit CPUs. Half-fit eliminates the search on free lists for allocation by calculating the index i and taking a free block of memory from the list i. If the list i is empty, the allocation algorithm takes a block from the subsequent non-empty list whose index is closest to i. If allocated blocks are larger than request sizes, the allocated block is split in two parts: one is returned to the requester and another one is relinked on the free list. The allocation/deallocation time complexity is O(1) [Ogasawara 1995].

Two-Level Segregated Fit memory allocator (TLSF) uses a segregated fit mechanism and implements a good-fit policy [Masmano et al. 2004]. TLSF limits the size of memory to be allocated in 16 bytes. It also uses the space of free blocks to store management information, such as pointers to next blocks. TLSF uses an array of free lists, which

each list (accessed by a specific array position) holds free blocks within a size class. The array of lists is organized in two-levels to speed up the access to free blocks and to reduce fragmentation. The first-level divides free blocks in classes that are power of two (16, 32, 64, 128, etc) and the second-level sub-divides each first-level class linearly, where the number of divisions is a user configurable parameter. As in Half-fit, a bitmap is used to identify empty lists and lists with free blocks. The boundary tag, used by TLSF, adds a pointer to the beginning of the same block to each free or used block. Thus, this pointer is used to locate the previous physical memory block and free or merge it when a block is released [Masmano et al. 2004]. Hence, each free block is linked in the segregated list and in a list ordered by physical address. Both allocation and deallocation have time complexity of O(1) [Masmano et al. 2004].

Sun *et al.* proposed the TLSF-I, an improved version of the TLSF algorithm [Sun et al. 2007]. TLSF-I aims at improving the efficiency in allocating small blocks and reduce fragmentation of the original TLSF algorithm. The authors used different strategies to allocate blocks of memory depending on their sizes. When the size of a block is small, TLSF-I uses exact-fit policy, instead of good-fit. To maintain an exact fit table, TLSF-I uses a two-level occupancy bitmap. When a memory block to be allocated has a small size, the algorithm uses the bitmap to to verify if that block fits in a free list. If it does not, the algorithm searches in the bitmap tree to find an available block. Comparative results have shown that TLSF-I has a better performance than TLSF and presented lower fragmentation [Sun et al. 2007].

Compact-FIT (CF) is a predictable memory allocation/deallocation system [Craciunas et al. 2008]. CF has a constant memory fragmentation rate and a linear response time, considering the size of a memory allocation request. Available memory in CF is partitioned in 16 KB pages. Then, pages are further partitioned into same-sized page-blocks class. When an allocation request arrives, the algorithm searches the page of the smallest-size class that is able to serve the request. The allocation of data larger than 16 KB is not supported. CF aims at compacting the memory size-class whenever possible, allowing at most one page of each size-class to be not-full at a given moment. When data is released, it is moved to keep the consistency in the size-class [Craciunas et al. 2008]. There are two implementation of CF: "in the moving CF implementation, page-blocks are mapped directly to physically contiguous pieces of memory, which requires moving data memory for compaction. In this implementation, allocation takes constant time and deallocation takes linear time if compaction occurs. In the non-moving implementation, a block table is used to map page-blocks into physical block-frames that can be located anywhere in memory. Compaction in this case is performed by reprogramming the block table rather than moving data" [Craciunas et al. 2008]. Both allocation and deallocation in the non-moving implementation take liner time. The authors compared CF with TLSF in terms of fragmentation. CF has presented a more controlled and predictable fragmentation than TLSF. In terms of allocation/deallocation time, Half-fit and TSLF are faster than CF due to compaction activities.

A completely different approach for time analysis of dynamic memory allocation is proposed by Herter and Reineke [Herter and Reineke 2009]. The authors propose algorithms that transform dynamic memory allocations into static memory, aiming at minimizing the WCET in a subsequent WCET analyses [Herter and Reineke 2009]. In order to allow the transformation from dynamic to static allocation, assumptions are made. For instance, loop bounds and requested block sizes must be statically known [Herter et al. 2011]. Good performance is only achieved when dynamic allocations are statically derived, which may not be true for every application or may be limited by the hardware platform (*i.e.,* when it has limited memory).

Puaut presented a performance analysis of several general purpose memory allocators (first-fit, best-fit, btree-best-fit, fast-fit, quick-fit, buddy-bin, and buddy-fibo) with

respect to real-time requirements [Puaut 2002]. By comparing the worst-case performance of allocation and deallocation of memory blocks in two scenarios (analyically and running real and synthetic workloads), the author has concluded that "for applications with low allocation rates, the analytically worst-case times do not have an excessive impact on the application execution times for the most predictable allocators (buddy systems and quick-fit)" [Puaut 2002].

More recently, Masmano *et al.* compared the first-fit, best-fit, binary-buddy [Knuth 1997], DLmalloc [Lea, D. 1996], Half-fit, and TLSF memory allocators for real-time applications [Masmano et al. 2006]. Results have indicated that TLSF and Half-fit can be used by real-time applications due to stable and bounded response times. In contrast, algorithms designed to optimize AVG execution times, such as DLmalloc and binary-buddy, are not suitable for real-time applications. Half-fit achieves bounded response times wasting more memory than TLSF [Masmano et al. 2006].

6.3. Cache-aware and time-predictable allocators

Cache-Aware Memory Allocator (CAMA) is the first memory allocator that combines constant time (predictability) with cache-awareness [Herter et al. 2011]. Constant search times are obtained by managing free blocks in segregated free lists. Consequently, CAMA provides constant response times, which is the key for predictability. A multi-layered segregated-list reduces the internal fragmentation. This segregatted list is similar to that proposed by TLSF [Herter et al. 2011]. Cache-awareness is obtained by adding an additional parameter to allocation requests: the first parameter is the original requested block size and the second and new parameter is the cache set in which the allocated memory block should be mapped to. Memory blocks within same size are kept in a single segregated list. Moreover, memory addresses that map to the same cache set are also grouped to ease cache-aware allocation. A bit vector for each cache set is responsible for signalizing that a block is empty or not. CAMA uses descriptor blocks to be aware of cache sets that are accessed during its execution. The idea is to insert these descriptor blocks in the segregated lists, indicating free memory blocks. Then, each memory block that is effectively allocated stores a pointer to its descriptor block [Herter et al. 2011].

6.4. Summary

Table V summarizes the OS memory allocators that are time-predictable or/and cache-aware. ccmalloc and CIF do not focus on real-time systems, although support page coloring and index-based cache management techniques, respectively. The dynamic to static allocation translation has the disadvantage of requiring additional assumptions about the program's allocation behavior, such as the size of memory allocation requests and loop bounds. The proposed C++ new operator overload technique does not provide a worst-case timing behavior analysis, but is the first work to provide page coloring allocation for internal OS memory pages. PALLOC modifies the Linux cgroup interface to provide colored memory allocation. Half-fit, TLSF, and TLSF-I provide a bounded time for allocation and deallocation activities. TLSF has lower fragmentation than Half-fit. TLSF-I improves the TLSF allocation for small memory blocks and reduces its fragmentation. CF was designed to have a low fragmentation rate due to an implemented compaction mechanism. Thus, CF has lower fragmentation than TLSF and Half-fit. However, CF is slower than TLSF and Half-fit for allocation/deallocation tasks. Finally, CAMA is the only memory allocator that provides a cache memory technique together with bounded allocation/deallocation time behavior.

Table V. Comparative table of OS memory allocators.

Memory Allocator	Time-predictable	Cache-aware	Cache management technique
ccmaloc [Chilimbi et al. 2000]	no	yes	clustering and page coloring
CIF [Afek et al. 2011]	no	yes	index-based
Dynamic translation [Herter and Reineke 2009]	yes	no	-
C++ new overload [Gracioli and Fröhlich 2013]	no	yes	page coloring
PALLOC [Yun et al. 2014]	no	yes	page coloring
Half-fit [Ogasawara 1995]	yes	no	-
TLSF [Masmano et al. 2004]	yes	no	-
TLSF-I [Sun et al. 2007]	yes	no	-
CF [Craciunas et al. 2008]	yes	no	-
CAMA [Herter et al. 2011]	yes	yes	index-based

7. FUTURE DIRECTIONS

None of the reviewed works support the three cache management mechanisms (cache locking or partitioning, and memory allocator). Some of them, such as [Suhendra and Mitra 2008] and [Mancuso et al. 2013], provide support for cache partitioning and locking, but are not integrated in a cache-aware real-time memory allocator. Other works, such as [Gracioli and Fröhlich 2013] and [Herter et al. 2011], have a cache-aware memory allocator, but do not support cache locking. Finally, the work in [Mancuso et al. 2015] integrates cache management using coloring and locking with a DRAM-aware memory allocator. The allocator, however, is not fully time-predictable. In general, the integration of the presented techniques is feasible and would result in a good approach to improve the predictability at the shared cache level.

Nevertheless, there are several processor architectural aspects that impact the predictability of real-time applications and are still open problems. Below, we provide a discussion of future research directions considering some aspects of current multicore processors, such as hardware prefetchers, I/O devices, cache coherence protocols, processor architecture, and random caches.

— **Hardware prefetchers:** By fetching instructions and/or data from memory to cache before the processor needs it, hardware prefetching reduces the latency of accesses to main memory. However, hardware prefetchers can be quite unpredictable. For instance, stride prefetchers use the distance (*i.e.,* stride of the load) between the current memory and last memory addresses referenced by a load instruction to fetch an address formed by the last address plus the stride distance. For a complete review on hardware prefetchers, please refer to [Lee et al. 2012].
Usually, real-time application designers disable hardware prefetchers to improve predictability. If page coloring could also be supported by hardware prefetchers, a real-time tasks would not interfere with another one. Moreover, rearranging data structures to be contiguously allocated in memory and re-using memory locations that have been released help to improve the hardware prefetcher performance.
— **Input and Output:** I/O devices can also interfere with real-time tasks at the shared cache level, either directly by DMA or through the cache coherence protocol (snooping). This behavior is the same as a regular core. Thus, cache locking and partitioning could also include I/O devices to avoid this interference. We believe that I/O is a hot future topic for real-time research.
— **Cache Coherence:** To the best of our knowledge no existing static WCET analysis technique is able to account for the effects of the coherence protocol. When tasks

share data or colors (in page coloring), they access the same cache lines, incurring in extra overhead. If frequent shared memory accesses occur, the WCET can be largely affected [Gracioli and Fröhlich 2013]. Depending on the application, shared data is intrinsic and cannot be avoided. In these cases, an alternative to decrease the contention caused by the cache coherence protocol is to combine page coloring, cache locking, scheduling, and hardware performance counters (HPCs). For instance, HPCs would feed the RTOS scheduler with run-time information about cache coherence activities, indicating when tasks are interfering with each other. Then, the scheduler can take an action, as to prevent the execution of interfering tasks. Another alternative is to disable caching for shared pages. Thus, invalidations caused by the coherence protocols are avoided.

— **Processor architecture:** Several features could be added to current multicore architectures to improve their predictability at the cache levels. For instance, to deal with the limited number of colors in page coloring, it would be possible to implement a mechanism of page recoloring. Without any additional hardware support, page recoloring requires the copy of complete pages, which may take a long time and lead to deadline misses. Frequent recoloring of a large number of tasks negates the benefit of page coloring [Zhang et al. 2009]. An approach is to provide a mechanism of re-tagging in the page address layout to enable page recoloring without page copies. Moreover, to provide HRT guarantees, cache locking is essential and must be supported by all embedded multicore processors. We suggest that hardware designers should add cache locking capabilities in future multicore processors. As cache locking is implemented in hardware, the run-time overhead is almost negligible.

Table VI provides a non-exhaustive list of COTS processors and platforms that feature hardware support for cache managements schemes (locking and partitioning). ARM9 single-core processors are among the first to introduce support for cache lockdown features in hardware. Dedicated registers are provided for controlling which cache ways are made unavailable for allocation to the standard replacement policy. However, ARM9 implementations often feature one level of virtually indexed, virtually tagged cache where locked data is flushed upon page-table switching.

Table VI. COTS CPUs and platforms supporting cache management features.

	Locking		Partition	
CPU Family	by Line	by Way	by Way	Example Platforms
ARM9	✓			Freescale i.MX233, Samsung S3C2440/S3C2510A
ARM Cortex-A9	✓	✓	✓	TI OMAP 4430/4460, NVIDIA Tegra 2/3/4i, Samsung Exynos 4210/4212/4412, Freescale i.MX 6 Solo-Quad, PandaBoard, ZedBoard
Freescale e500/e500mc	✓		(✓)	Freescale P40xx-series, Freescale MPC85xx
Freescale e600/e600mc	✓		(✓)	Freescale MPC86xx
Freescale e5500/e5500mc	✓		✓	Freescale P5020/P5010
Freescale e6500/e6500mc	✓		✓	Freescale T4xxx-series
Intel Xeon E5 v3			✓	Intel Xeon E5-16xx v3, Intel Xeon E5-26xx v3

ARM Cortex-A9 CPUs introduce a second level of cache which is shared in multicore implementations and managed by the L2C-310 (PL-213) cache controller. L2 sizes can range from 512 KB to 2 MB in these chips and cache blocks are physically indexed and tagged. Although cache locking features can be configured at synthesis time by manufacturers, it is often the case that L2 controllers in Cortex-A9 chips are

configured to support cache management features. A non-atomic lockdown by-line scheme is provided, together with the more flexible lockdown by-master. In the latter scheme, each core is associated a register to configure the state of each cache way with respect to operations performed by the considered core. For instance, the same cache way can be configured to accept allocations for lines fetched by core A, while not allowing any allocation for requests originated from core B. As such, lockdown by-master can be used to enforce traditional by-way locking or by-way partitioning.
Freescale PowerPC platforms have historically provided more attention to safety-critical applications, often adding advanced support for performance monitoring and tuning. As a part of the extended hardware support offered in these platforms, cache management schemes are often provided at the ISA level or at the cache controller level. Freescale e500 and e600 processors feature 2 levels of private cache. In both single-core and multi-core configurations (e500mc and e600mc), they provide atomic instructions to perform by-line prefetch and lock of cache content. These instructions can be used to manage the content of both the private levels of cache. Multi-core configurations introduce a third level of cache, called platform cache, which can be managed using the same instructions and feature additional by-way/by-master cache partitioning capabilities. The newer Freescale e5500 and e6500 CPUs, not only maintain the support for atomic cache management instructions, but also introduce by-thread cache partitioning capabilities at the second level of private cache (L2).
Finally, Intel has recently introduced support for cache management and monitoring on server-grade CPUs: the Xeon E5 v3 family. Although these processors are not meant for the embedded market, soft real-time applications can be optimized at an OS level to use the large caches (20 MB to 45 MB) of these processors through the newly introduced features. The Intel Cache Allocation Technology (CAT) allows the definition of classes of service (COS). Each thread can be associated to a COS and for each COS the hardware provides a cache allocation bitmap. If bit n is set in the bitmap associated to COS A, all the threads defined in this class of service will be allowed to allocate cache lines in the $n - th$ cache partition. Bitmaps for different COS's can be configured independently, allowing de-facto overlapped and private assignment of cache blocks to threads.

— **Random Caches:** Recently, PTA (Probabilistic Timing Analysis) has been proposed as an alternative to conventional timing analysis that can be highly pessimistic for the worst-case [Cazorla et al. 2013]. PTA provides pWCET estimates (Probabilistic WCET) that can be exceeded with a given probability. That is, a pWCET with an associated low probability, say 10^{-15}, means that the probability for the execution time to exceed this pWCET is 10^{-15}. However, PTA techniques require the execution times of programs to have a probability of occurrence that is independent and identically distributed. These two features are essential to allow using random variables and apply statistical methods for analyzing the system. At the cache level, caches with LRU replacement policy cannot be used because the result of each memory access is dependent on the previous accesses. A fully-associate cache with random replacement is one example that can be used for PTA. Along this line of research, PTA has been applied to a single level cache [Kosmidis et al. 2013a], multi-level caches [Kosmidis et al. 2013b] and for CRPD [Davis et al. 2013]. Recently, PTA was used for shared caches to estimate the inter-core cache conflicts [Slijepcevic et al. 2014].

— **Translation Lookaside Buffer (TLB):** The paging mechanism relies on a table to perform address translation at run-time. Since the table is too large to fit in processor registers, instead a subset of the table is stored in a hardware structure known as TLB. The TLB functions as a specialized data cache for page table entries, and therefore suffers from predictability problems common to other caches, including intra-task, intra-core and inter-core interference [Panchamukhi and Mueller 2015]. Due

to its similarity to data caches, both locking and partitioning solutions appear to be feasible, but there is very little published work focusing on TLB management for real-time systems: [Ishikawa et al. 2013] discusses TLB locking based on the support provided by some ARM platforms, while [Panchamukhi and Mueller 2015] supports index-based partitioning in the TLB using page coloring.

— **Profiling Techniques:** The majority of challenges in designing cache management techniques lay in the establishment of a good trade-off between the determinism imposed on the hardware circuitry and the resulting performance. However, Without a full knowledge of the application behavior in terms of memory access pattern, this trade-off is decided empirically for all the applications. Consequently, only specific classes of applications can benefit from a specific cache management technique. The ability to profile and analyze the key aspects of applications' memory access patterns is a turning point in the optimization of cache management techniques. A body of works exists that use abstract interpretation [Cousot 2001; Cullmann 2013], symbolic execution [King 1976], hardware and software memory tracing [Sun and Tian 2011; Xiaofeng et al. 2005; Cesati et al. 2015] to gather information about memory access patterns. These techniques have been mostly used to perform static analysis. Nonetheless, similar approaches could be used to evolve cache management mechanism into memory-pattern-aware techniques.

8. CONCLUSION

The migration to multicore architectures is posing incredible challenges to the development of hard real-time systems. This is because the presence of shared hardware resources, such as memories, interconnects, I/O, etc., creates undue interferences among cores. In turn, this interference makes it incredibly hard to derive safe and accurate worst-case bounds on tasks' execution times, which is required to guarantee timing constraints (deadline). As part of this survey paper, we have specifically focused on analysis of CPU cache hierarchy. We have introduced and classified the various types of cache interference, and argued that mitigation strategies ought to be implemented to provide isolation among tasks and cores. We have classified available cache management mechanisms into either index-based cache partitioning, way-based cache partitioning, cache locking, and OS memory allocators, and described techniques proposed in the real-time literature for each category. Different techniques have been compared based on required hardware support and provided isolation guarantees.

A main take-away of our survey is that a certain level of hardware support is necessary to provide strong isolation guarantees to concurrently executing tasks in a multicore real-time system. Even when software-only implementation is possible, such solutions are typically more cumbersome, more difficult to certify and/or add more overhead compared to hardware solutions. For example, index-based partitioning can be achieved through compiler-based techniques, but modifying and re-certifying the compiler can be a daunting task. Furthermore, while the present work focuses on cache memories, modern multicore processors include other shared resources which exhibit similar behavior to caches, that is, they are highly stateful components where past interactions by other cores can greatly influence the latency of future accesses. Hence, we suggest that architectural design for hard real-time systems should focus on providing isolation properties at the hardware level.

References

Y. Afek, D. Dice, and A. Morrison. 2011. Cache index-aware memory allocation. In *Proc. of the ISMM*. ACM, USA, 55–64.

AMD. 2013. AMD64 Architecture Programmers Manual Volume 2: System Programming. Section 7.3: Memory Coherency and Protocol. (May 2013.). Publication # 24593. Revision: 3.23.

L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals. 2011. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *Journal of Systems Architecture* 57, 7 (2011), 695–706.

Alexis Arnaud and Isabelle Puaut. 2006. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the 14th RTNS*.

Abu Asaduzzaman, Fadi N. Sibai, and Manira Rani. 2010. Improving cache locking performance of modern embedded systems via the addition of a miss table at the {L2} cache level. *Journal of Systems Architecture* 56, 4-6 (2010), 151–162.

Mathieu Avila and Isabelle Puaut. 2009. Heptane - A Tree-based WCET Analysis Tool. (2009). http://ralyx.inria.fr/2004/Raweb/aces/uid43.html

Andrea Bastoni, Björn Brandenburg, and James Anderson. 2010. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *Proc. of OSPERT*. 33–44.

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of Linux scalability to many cores. In *Proc. of the 9th OSDI*. USENIX Association, Berkeley, CA, USA, 1–8.

Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. 1996. Compiler-directed Page Coloring for Multiprocessors. In *Proc. of the 7th ASPLOS*. ACM, USA, 244–255.

B.D. Bui, M. Caccamo, Lui Sha, and J. Martinez. 2008. Impact of Cache Partitioning on Multi-tasking Real Time Embedded Systems. In *Proc. of the 14th RTCSA*. 101–110.

Marti Campoy, A. Perles Ivars, and J. V. Busquets Mataix. 2001. Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. In *Proc. of IEEE Real-Time Embedded Systems Workshop*. London, UK.

Francisco J Cazorla, Eduardo Quinones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, and others. 2013. Proartis: Probabilistically analyzable real-time systems. *ACM TECS* 12, 2s (2013), 94.

Certification Authorities Software Team (CAST). 2014. Position Paper on Multi-core Processors - CAST-32. (May 2014). https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf

M. Cesati, R. Mancuso, E. Betti, and M. Caccamo. 2015. A Memory Access Detection Methodology for Accurate Workload Characterization. In *Proc. of the 21th IEEE RTCSA*. IEEE.

Yu Chen, Wenlong Li, Changkyu Kim, and Zhizhong Tang. 2009. Efficient shared cache management through sharing-aware replacement and streaming-aware insertion policy. In *Proc. of the 24th IEEE IPDPS 09*. 1–11.

T. M. Chilimbi, M. D. Hill, and J. R. Larus. 2000. Making Pointer-Based Data Structures Cache Conscious. *Computer* 33, 12 (Dec. 2000), 67–74.

Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. 2000. Dynamic cache partitioning via columnization. In *Proc. of DAC*. ACM.

Ali Chousein and Rabi N. Mahapatra. 2005. Fully associative cache partitioning with don't care bits for real-time applications. *SIGBED Rev.* 2, 2 (April 2005), 35–38.

P. Cousot. 2001. Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics - 10 Years Back. 10 Years Ahead.* Springer-Verlag, London, UK, UK, 138–156.

Silviu S. Craciunas, Christoph M. Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger. 2008. A Compacting Real-time Memory Management System. In *Proc. of USENIX ATC*. USENIX, USA, 349–362.

C. Cullmann. 2013. Cache Persistence Analysis: Theory and Practice. *ACM Trans. Embed. Comput. Syst.* 12, 1s, Article 40 (March 2013), 25 pages.

C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, S. Wegener, and R. Wilhelm. 2010. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingénieurs de l'Automobile* 807 (September 2010), 36–42.

Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *Comput. Surveys* 43, 4, Article 35 (Oct 2011), 44 pages.

R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. 2013. Analysis of probabilistic cache related pre-emption delays. In *Proc. of the 25th ECRTS*. IEEE, 168–179.

Huping Ding, Yun Liang, and T. Mitra. 2012. WCET-centric partial instruction cache locking. In *Proc. of the 49th ACM/IEEE DAC*. 412–420.

Huping Ding, Yun Liang, and T. Mitra. 2014. WCET-Centric dynamic instruction cache locking. In *Proc. of DATE*. 1–6.

Heiko Falk, Sascha Plazar, and Henrik Theiling. 2007. Compile-time Decided Instruction Cache Locking Using Worst-case Execution Paths. In *Proc. of the 5th IEEE/ACM CODES+ISSS*. ACM, USA, 143–148.

C. Ferdinand. 1997. *Cache Behavior Prediction for Real-Time Systems*. Ph.D. Dissertation. Saarland University.

G Gracioli and A. A. Fröhlich. 2013. An Experimental Evaluation of the Cache Partitioning Impact on Multicore Real-Time Schedulers. In *Proc. of the 19th IEEE RTCAS*. IEEE, 10.

Daniel Grund and Jan Reineke. 2010a. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *Proc. of the 22nd ECRTS*. 155–164.

Daniel Grund and Jan Reineke. 2010b. Toward Precise PLRU Cache Analysis. In *Proc. of 10th Workshop on WCET Analysis*. 28–39.

Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. 2009. Cache-aware scheduling and analysis for multicores. In *Proc. of the EMSOFT'09*. ACM, 245–254.

Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. 2013. FIFO cache analysis for WCET estimation: a quantitative approach. In *Proc. of DATE*. USA, 296–301.

Damien Hardy, Thomas Piquet, and Isabelle Puaut. 2009. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proc. of the 30th RTSS*. IEEE, 68–77.

Damien Hardy and Isabelle Puaut. 2009. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proc. of the 17th RTNS*. 45–54.

John L. Hennessy and David A. Patterson. 2006. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, Fourth Edition.

J. Herter, P. Backes, F. Haupenthal, and J. Reineke. 2011. CAMA: A Predictable Cache-Aware Memory Allocator. In *Proc. of the 2011 ECRTS*. 23 –32.

J. Herter and J. Reineke. 2009. Making dynamic memory allocation static to support WCET analyses. In *Proc. of 9th International Workshop on WCET Analysis*.

Intel. 2010. An introduction to the Intel QuickPath Interconnect. (January 2010).

T Ishikawa, T Kato, S Honda, and H Takada. 2013. Investigation and improvement on the impact of TLB misses in real-time systems. In *Proc. of OSPERT*.

Ravi Iyer. 2004. CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms. In *Proc. of the 18th ICS*. ACM, USA, 257–266.

R. E. Kessler and Mark D. Hill. 1992. Page Placement Algorithms for Large Real-indexed Caches. *ACM Trans. on Computer Systems* 10, 4 (Nov 1992), 338–359.

Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. 2013. A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems. In *Proc. of the 25th ECRTS*. 80–89.

J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394.

D.B. Kirk and J.K. Strosnider. 1990. SMART (strategic memory allocation for real-time) cache design using the MIPS R3000. In *Proc. of the 11th RTSS*. 322–330.

Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J Cazorla. 2013a. A cache design for probabilistically analysable real-time systems. In *Proc. of the DATE*. 513–518.

L. Kosmidis, J. Abella, R. Quinones, and F. J. Cazorla. 2013b. Multi-level unified caches for probabilistically time analysable real-time systems. In *Proc. of the 34th RTSS*. IEEE, 360–371.

Lea, D. 1996. A Memory Allocator. (1996). Unix/Mail, 6/96.

Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. 2012. When Prefetching Works, When It Doesn&Rsquo;T, and Why. *ACM Trans. Archit. Code Optim.* 9, 1, Article 2 (March 2012), 29 pages.

Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems* 48, 6 (2012), 638–680.

Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. 1997. OS-Controlled Cache Predictability for Real-Time Systems. In *Proc. of the 3rd IEEE RTAS*. IEEE, USA, 213–224.

Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proc. of the HPCA*. IEEE, 367–378.

Chun Liu, Anand Sivasubramaniam, and Mahmut Kandemir. 2004. Organizing the Last Line of Defense before Hitting the Memory Wall for CMPs. In *Proc. of the 10th HPCA*. IEEE, USA, 176–.

C. L. Liu and J. W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (Jan. 1973), 46–61.

Tiantian Liu, Minming Li, and C.J. Xue. 2009a. Instruction Cache Locking for Real-Time Embedded Systems with Multi-tasks. In *Proc. of the 15th IEEE RTCSA*. 494–499.

Tiantian Liu, Minming Li, and C.J. Xue. 2009b. Minimizing WCET for Real-Time Embedded Systems via Static Instruction Cache Locking. In *Proc. of the 15th IEEE RTAS*. 35–44.

Qingda Lu, Jiang Lin, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2009. Soft-OLP: Improving Hardware Cache Performance through Software-Controlled Object-Level Partitioning. In *Proc. of the 18th PACT*. 246–257.

R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. 2013. Real-time cache management framework for multi-core architectures. In *Proc. of the 19th IEEE RTAS*. USA, 45–54.

Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. 2015. WCET(m) Estimation in Multi-Core Systems using Single Core Equivalence. In *Proc. of the 27th ECRTS*.

Miguel Masmano, Ismael Ripoll, and Alfons Crespo. 2006. A Comparison of Memory Allocators for Real-time Applications. In *Proc. of the 4th JTRES*. ACM, USA, 68–76.

M. Masmano, I. Ripoll, A. Crespo, and J. Real. 2004. TLSF: a new dynamic memory allocator for real-time systems. In *Proc. of the 16th ECRTS*. 79–88.

S. Mohan, M. Caccamo, L. Sha, R. Pellizzoni, G. Arundale, R. Kegley, and D. de Niz. 2011. Using Multicore Architectures in Cyber-Physical Systems. In *Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components*. Michigan, USA.

Frank Mueller. 1995. Compiler Support for Software-based Cache Partitioning. In *Proc. of the ACM SIGPLAN LCTES*. ACM, USA, 125–133.

S.P. Muralidhara, M. Kandemir, and P. Raghavan. 2010. Intra-application cache partitioning. In *Proc. of the 25th IEEE IPDPS*. 1–12.

T. Ogasawara. 1995. An Algorithm with Constant Execution Time for Dynamic Storage Allocation. In *Proc. of the 2Nd RTCSA*. IEEE, USA, 21–.

S Panchamukhi and F Mueller. 2015. Providing Task Isolation Via TLB Coloring. In *Proc. of the 21th RTAS*.

Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. 2009. Hardware support for WCET analysis of hard real-time multicore systems. In *ACM SIGARCH Computer Architecture News*, Vol. 37. ACM, 57–68.

R. Pellizzoni. 2010. *Predictable and Monitored Execution for COTS-based Real-Time Embedded Systems*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

I. Puaut. 2002. Real-time performance of dynamic memory allocation algorithms. In *Proc. of the 14th ECRTS*. 41–49.

I. Puaut and D. Decotigny. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proc. of the 23rd IEEE RTSS*. 114–123.

I Puaut and C Pais. 2007. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. of DATE*. 1–6.

Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of MICRO 39*. IEEE, 423–432.

Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. 2006. Architectural Support for Operating System-driven CMP Cache Management. In *Proc. of the 15th PACT*. ACM, USA, 2–12.

Parthasarathy Ranganathan, Sarita Adve, and Norman P. Jouppi. 2000. Reconfigurable Caches and Their Application to Media Processing. In *Proc. of the 27th ISCA*. ACM, USA, 214–224.

Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing Predictability of Cache Replacement Policies. *Real-Time Systems* 37, 2 (November 2007), 99–122.

Theodore Romer, Dennis Lee, Brian N. Bershad, and J. Bradley Chen. 1994. Dynamic Page Mapping Policies for Cache Conflict Resolution on Standard Hardware. In *Proc. of the 1st OSDI*. 255–266.

Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. 2011. Predictable task migration for locked caches in multi-core systems. In *Proc. of the LCTES'11*. ACM, New York, 131–140.

Abhik Sarkar, Frank Mueller, and Harini Ramaprasad. 2012. Static task partitioning for locked caches in multi-core real-time systems. In *Proc. of the CASES '12*. ACM, NY, USA, 161–170.

Timothy Sherwood, Brad Calder, and Joel Emer. 1999. Reducing Cache Misses Using Hardware and Software Page Placement. In *Proc. of the 13th ICS*. ACM, USA, 155–164.

Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. 2014. Time-analysable non-partitioned shared caches for real-time multicore systems. In *Proc. of the 51st ACM/IEEE DAC*. IEEE, 1–6.

S. Srikantaiah, M. Kandemir, and M. J. Irwin. 2008. Adaptive set pinning: managing shared caches in chip multiprocessors. In *Proc. of the 13th ASPLOS*. ACM, 135–144.

G. E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic Partitioning of Shared Cache Memory. *The Journal of Supercomputing* 28, 1 (Apr 2004), 7–26.

Vivy Suhendra and Tulika Mitra. 2008. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proc. of the 45thDAC*. ACM, USA, 300–303.

Q. Sun and H. Tian. 2011. A flexible automatic source-level instrumentation framework for dynamic program analysis. In *Proc. of the 2nd IEEE ICSESS*. 401 –404.

Xiao Hui Sun, JinLin Wang, and Xiao Chen. 2007. An Improvement of TLSF Algorithm. In *Proc. of the 15th IEEE-NPSS*. 1–5.

K.T. Sundararajan, T.M. Jones, and N.P. Topham. 2013. RECAP: Region-Aware Cache Partitioning. In *Proc. of the 31st IEEE ICCD*. 294–301.

D. Tam, R. Azimi, L. Soares, and M. Stumm. 2007. Managing shared L2 caches on multicore systems in software. In *Proc. of the WIOSCA*.

George Taylor, Peter Davies, and Michael Farmwald. 1990. The TLB Slice - a Low-cost High-speed Address Translation Mechanism. In *Proc. of the 17th ISCA*. ACM, New York, NY, USA, 355–363.

K. Varadarajan, S. K. Nandy, V. Sharda, A. Bharadwaj, R. Iyer, S. Makineni, and D. Newell. 2006. Molecular Caches: A caching structure for dynamic creation of application-specific Heterogeneous cache regions. In *Proc. of the 39th MICRO*. 433–442.

Xavier Vera, Björn Lisper, and Jingling Xue. 2003a. Data Cache Locking for Higher Program Predictability. In *Proc. of ACM SIGMETRICS*. ACM, New York, NY, USA, 272–282.

Xavier Vera, Björn Lisper, and Jingling Xue. 2003b. Data Caches in Multitasking Hard Real-Time Systems. In *Proc. of the RTSS'03*. IEEE, 154–.

B.C. Ward, J.L. Herman, C.J. Kenna, and J.H. Anderson. 2013. Making Shared Caches More Predictable on Multicore Platforms. In *Proc. of the 25th ECRTS*. 157–167.

J Whitham and N Audsley. 2009. Implementing time-predictable load and store operations. In *Proc. EMSOFT*. 265–274.

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. 2008. The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools. *ACM TECS* 7, 3, Article 36 (May 2008), 53 pages.

Andrew Wolfe. 1994. Software-based Cache Partitioning for Real-time Applications. *Journal of Computing Software Engineering* 2, 3 (March 1994), 315–327.

G. Xiaofeng, M. Laurenzano, B. Simon, and A. Snavely. 2005. Reducing overheads for acquiring dynamic memory traces. In *Proc. of the IEEE IISWC*. 46–55.

Jun Yan and Wei Zhang. 2008. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *Proc. of the 14th IEEE RTAS*. 80–89.

H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. 2014. PALLOC: DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms. *Proc. of the 20th IEEE RTAS* (April 2014).

Wei Zhang and Yan Jun. 2012. Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering* 6, 4 (2012), 267–278.

Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proc. of the 4th ACM EuroSys*. ACM, USA, 89–102.

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *Comput. Surveys* 45, 1, Article 4 (Dec 2012), 28 pages.

Received X 2015; revised Y 2015; accepted Z 2015