Deterministic Memory Hierarchy and Virtualization for Modern Multi-Core Embedded Systems

Tomasz Kloda*, Marco Solieri*, Renato Mancuso†, Nicola Capodieci*, Paolo Valente*, and Marko Bertogna*
*Università di Modena e Reggio Emilia, Italy
{name.surname}@unimore.it
†Boston University, USA
rmancuso@bu.edu

***Abstract*—One of the main predictability bottlenecks of modern multi-core embedded systems is contention for access to shared memory resources. Partitioning and software-driven allocation of memory resources is an effective strategy to mitigate contention in the memory hierarchy. Unfortunately, however, many of the strategies adopted so far can have unforeseen side-effects when practically implemented latest-generation, high-performance embedded platforms. Predictability is further jeopardized by cache eviction policies based on random replacement, targeting average performance instead of timing determinism.**

In this paper, we present a framework of software-based techniques to restore memory access determinism in high-performance embedded systems. Our approach leverages OS-transparent and DMA-friendly cache coloring, in combination with an invalidation-driven allocation (IDA) technique. The proposed method allows protecting important cache blocks from (i) external eviction by tasks concurrently executing on different cores, and (ii) internal eviction by tasks running on the same core. A working implementation obtained by extending the Jailhouse partitioning hypervisor is presented and evaluated with a combination of synthetic and real benchmarks.

I. Introduction

New high-end embedded platforms are paving the way towards the execution of workload intensive applications in the cyber-physical domain. Autonomous vehicles, smart manufacturing systems and advanced embedded controllers are required to couple, onto the same computing platform, a variety of control tasks with vision and machine learning routines. The increasing request for computing power within tight real-time constraints calls for techniques for predictably mastering the impressive computing power made available by next-generation embedded multi-core architectures. Unfortunately, such a higher computing power is obtained at the expense of predictability. Modern architectures are based on host clusters where multiple cores are tightly coupled with the shared memory hierarchy, leading to contention problems that are detrimental to the worst-case performance of real-time activities. Architectural trends favour design decisions aiming at improving average performance, while exposing to particularly harmful corner cases that may significantly increase worst-case response times. Such features include: out-of-order execution, speculative branching, hardware prefetching, and dynamic code optimization (DCO) [4].

While a number of these features can be disabled, others need to be analyzed and accounted for when computing the worst-case execution time (WCET) of applications. An effective meet-in-the-middle approach consists in using software (e.g. the OS) to manage the behavior of the hardware. In this way, it is possible to avoid by construction particularly pessimistic scenarios, enabling tighter WCET calculation.

There is a large consensus that architectural features introducing non-determinism in the memory hierarchy represent a major source of unpredictability. The main performance and predictability bottlenecks are arguably represented by shared memory (and I/O) resources. Producing tight WCET bounds for applications that compete for access to shared memory resources is particularly problematic. In fact, the temporal behavior of a task under analysis is severely influenced by logically unrelated applications executing in parallel on different cores, as they compete with each other for access to memory resources.

An effective approach towards mitigating the unpredictability arising from a shared memory hierarchy is partitioning. While being a step in the right direction, partitioning alone is not sufficient to guarantee strict temporal determinism. Consider for instance a shared cache where the controller implements a pseudo-random replacement policy. Even without the effect of inter-core contention, WCET analysis has to pessimistically assume that every access to a new cache line will result in a miss.

Comparatively less work has explored leveraging platform-specific support to *override* the cache replacement policy so to deterministically allocate cache blocks to real-time applications. Unfortunately, however, vendors of modern embedded Systems-on-Chip (SoCs) are increasingly phasing out primitives for explicit cache content management (e.g. cache locking, cache stashing), making such techniques not directly applicable on latest-generation SoCs. On top of that, while some partitioning techniques (e.g. memory coloring) can still be applied in modern SoCs, their practical implementation has deep implications that have been only marginally explored.

In this paper, we consider the current trend in multi-core embedded platforms and assess what is the "common denominator" in terms of architectural features across different vendors. With this in mind, the purpose of this work is twofold. On the one hand, we discuss indirect effects of traditional memory partitioning, and propose mitigation strategies when strict partitioning is required. On the other hand, we propose a novel technique to perform deterministic cache content allocation by relying on a previously overlooked invariant of a generic cache controller.

In summary, this work makes the following contribution:

1) We describe an efficient implementation of memory coloring to perform both cache and DRAM bank partitioning by leveraging virtualization extensions in modern embed-

ded systems platforms;

2) We detail how coloring can be dynamically performed. This allows us to support systems where a main "partition" is booted without coloring (legacy mode), and coloring is later activated/deactivated if the system undergoes a mode change;
3) We discuss a number of overlooked side-effects of coloring and propose solutions to mitigate the issues. These include: reduction on the available memory space; use of DMAs and DMA-capable I/O devices; undesired partitioning of other cache levels; increased latency for page table walk;
4) We detail a technique to deterministically control the content of a cache with no support for locking and generic replacement policy – including random.

The remainder of the paper is organized as follows. Section II contains a brief survey of related work. Section III summarizes key architectural features of the considered SoCs. Section IV discusses the proposed approach for strict memory partitioning. A deterministic cache allocation strategy is then proposed in Section V, and evaluation considerations are made in Section VI. The paper concludes in Section VII.

II. Related Work

Unregulated space contention at the shared levels of cache in a multi-core environment represents a major source of temporal unpredictability for real-time tasks. This problem has been extensively studied in the literature. With specific focus on multi-core systems, two main approaches have emerged: cache partitioning and cache locking.

Cache partitioning strategies restrict cache line allocation from system entities, i.e., tasks or cores, to a subset of the available cache space. This can be achieved using specialized hardware platform support, or via careful software-level management of physical memory. Both approaches have advantages and disadvantages. On the one hand, cache partitioning performed using platform-specific support can be easily enforced at boot-time and requires limited changes at the Operating System (OS) level. A number of works has investigated the use of commercially available hardware support for cache partitioning [41], [25], [15]. A considerably larger body of works has proposed hardware modifications to partition the cache space [18], [7], [42], [22]. Compared to these works, we focus on commercially available platforms that do not provide any specific support for cache management. This is the case for the NVIDIA Tegra SoM used in our experiments.

Software approaches to perform partitioning have also been studied in the literature. These are usually implemented at compiler-level [29], [30], [6], with changes to the application [5], [32], or at the OS level [20], [47], [8], [44], [23]. These solutions limit the portability for cache partitioning mechanisms, because they require modifications to the toolchain, the OS, or both. Cache partitioning via page coloring [21], [45], [35], [25], [44], [34], [37], [23], [43] can be considered as a special case of software-enforced cache partitioning, typically implemented by modifying the behavior of the physical memory allocator in the OS.

Our work relies on page coloring. Differently from the literature mentioned above, however, we rely on virtualisation support. A lightweight hypervisor, namely Jailhouse, allocates physical memory pages that map to non-overlapping cache sets to different guest OS's. Page coloring implemented at the hypervisor level allows us to partition the last-level cache among different guest OS's, while requiring no modifications to OS-level memory allocators, and/or to the toolchain of the guest OS's. In this sense, our work is related to a number of recent techniques that leverage an implementation of page coloring at the hypervisor-level [13], [40], [28]. In comparison, our work sets itself apart because: (i) we propose a technique to perform deterministic allocation of cache content to lower pessimism in estimating Worst-Case-Execution Time (WCET); (ii) we conduct an in-depth evaluation on the impact of the additional memory translation layer introduced by virtualisation; and (iii) we investigate the benefits of cache invalidation, software-, and hardware-driven prefetching.

TABLE I: Features and spread of ARMv8 processors

	ARM A57	*ARM A53*
Cache Locking	No	No
L1 Placement	Mostly PIPT	Mostly VIPT
L2 Placement	Mostly PIPT	Mostly PIPT
L2 Replacement policy	Pseudo-random	Pseudo-random
L2 Size (per cluster)	512 KiB - 2 MiB	128 KiB - 2 MiB
L2 Associativity	16 ways	16 ways
L2 is LLC	Mostly Yes	Mostly Yes
Hardware Virtualization	Yes	Yes
Commercial platforms and SoCs	NVIDIA TX1 NVIDIA TX2 Hisilicon D02	NXP i.MX 8 series NXP BlueBox NVIDIA TK1 Xilinx Ultrascale+ MPSoC Intel Stratix 10 SoC Raspberry Pi 3 B+

Cache Locking has been traditionally studied as a way to construct guarantees on the worst-case number of hits/misses observed in real-time applications [25], [24], [31], [36], [3]. Cache locking inherently relies on platform-specific hardware support that allows one to specify which cache lines should be protected against eviction. Unfortunately, support for cache locking is not available in modern generations of embedded Systems on Module (SoMs). For instance, cache locking capabilities have been removed in the transition from ARM Cortex-A9 to ARM Cortex-A15. In this work, we demonstrate that it is possible to exert deterministic control over cache content regardless of the replacement policy implemented at the cache controller, and without hardware-specific support.

Other approaches that aimed at deriving guarantees on the content of cache lines to predict hits/misses focused on deterministic replacement policy, such as Least Recently Used (LRU) [46], [9]. Conversely, the work presented hereby can be used in spite of non-deterministic eviction strategies at the cache controller, such as pseudo-random replacement policy. It is important to underline that pseudo-random replacement is implemented in the vast majority of commercial high-performance embedded SoMs, including the NVIDIA Tegra X1 on which we base our evaluation.

III. Reference Architecture

In this section, we introduce the key architectural features that are leveraged, and assumed available, on the considered high-end embedded SoCs. We also provide basic background notions required to better understand this work. Notable examples of widely adopted commercial platforms that are compatible with this work are provided in Table I.

A. Memory Addressing and Virtualization

We consider modern multi-core embedded SoCs that provide virtualization support in hardware. This is a common trend for latest-generation embedded platforms intended for high-performance applications. Memory is organized as follows. First, there exists a single physical addressing space where main memory (e.g. DRAM) as well as I/O devices are mapped. In other words, any memory-mapped hardware resource is reachable under a range (aperture) of physical addresses (PA). Second, two stages of memory virtualization are supported by the hardware. The first stage corresponds to the typical infrastructure used to implement multi-programming. At this stage, virtual addresses (VA) seen by user-space processes are translated into intermediate physical addresses (IPA) by the Memory Management Unit (MMU). The page tables with the mapping between VA and IPA are entirely managed by an OS.

The second stage of translation maps IPAs as seen by the OS into PAs. This mapping in managed by a hypervisor, which operates at a higher privilege level compared to the OS. If second-stage translation is disabled – for instance because no hypervisor is active in the system – then an IPA translates directly and 1-to-1 to a PA. Conversely, if a hypervisor is enabled, it is responsibility of the hypervisor to setup appropriate second-stage page tables to allow translation of IPA to PA. The same circuitry – i.e. the MMU – that handles first-stage translations is used to perform second-stage translation for memory accesses originated by the CPUs.

In summary, support for two-stage memory address translation provides a hypervisor (if enabled) the "last word" on which physical addresses (and hence devices) will be effectively accessible from applications and virtualized OS's. In the vast majority of platforms, the finest granularity at which VA, IPA and PA can be managed is 4 KiB.

B. Multi-level Caches

Modern embedded SoCs may feature multiple levels of caches with complex hierarchies. A widely adopted model, however, is the following. Each core features a first, private cache level (L1) which is kept coherent with other core's private caches. Additionally, all the cores share a second level of cache, namely L2. The L2 cache is typically much larger in space compared to private L1 caches. We consider platforms where L2 is also the last-level cache (LLC), but our considerations may be easily extended to next-generation embedded platforms where a third cache level is available (e.g. Nvidia Orin, Xilinx Versal, etc.). A miss in LLC causes an access to main memory.

1) Cache Structure: Caching at any level (L1 or L2) follows a *set-associative* scheme. A set-associative cache with associativity W features W *ways*. Each way has an identical structure. It follows that if the cache has size C_S, it will be structured in W ways of size $W_S = C_S/W$ each. The generic structure of a set-associative cache is depicted in Figure 1.

For efficiency reasons, caches do not store individual bytes. Instead they store multiple consecutive bytes at a time, which form a *cache line* (or *cache block*). We use L_S to indicate the number of bytes in each cache line. Typical sizes are 32 bytes or 64 bytes. The number of lines in a way is $S = W_S/L_S$, also called the number of *sets* of a cache. Each set contains W lines, one per way.

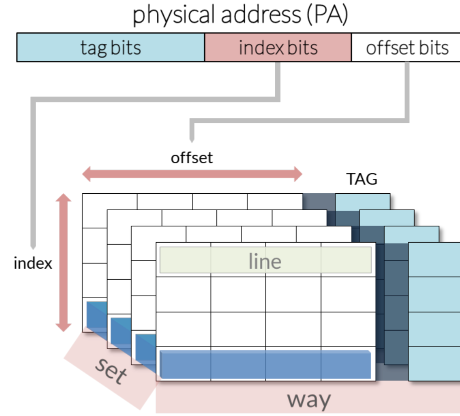


Fig. 1: Structure of a 4-way, 4-sets set-associative cache.

When a *cacheable* memory location is accessed by a CPU, the value of its address determines which cache location to look-up or allocate in case of cache miss. The least-significant bits of the address encode the specific byte inside the cache line, and thus do not affect where the line will be allocated. For instance, in systems where $L_S = 64$ bytes, these are the last $\log_2 L_S = 6$ bits (bits [5:0]) of a memory address. This group of bits is called *offset*. A second group of bits in the memory address encodes the specific cache set in which the memory content can be cached. Since we have S possible sets, the next $\log_2 S$ bits after the offset bits select one of the possible sets. These are called *index* bits. Finally, a memory address has more bits than the ones used as offset and index bits. The remaining bits, namely *tag* bits, are stored alongside with cached content to detect cache hits after look-up – see Figure 1.

Addresses used for cache look-ups can be PAs or a VAs. In the vast majority of embedded multi-core systems, tag bits are always PA bits – *physically-tagged* caches. In shared caches (e.g. L2/LLC), index bits are also from PAs. For this reason, they are said to be *physically-indexed*, *physically-tagged* (PIPT) caches. It is however not uncommon for private cache levels (e.g. L1) to operate with index bits from VAs. In this case, they are *virtually-indexed*, *physically-tagged* (VIPT). *This often overlooked distinction is fundamental when jointly partitioning cache space and DRAM banks.*

2) Cache Colors: Recall that an OS/hypervisor can manage VA/IPA/PA addresses at the granularity of 4 KiB pages. It follows that a page spans through multiple cache lines. For instance, each 4 KiB page will contain 128 cache lines if $L_S = 32$ bytes. Moreover, if the way size W_S is larger than 4 KiB, then multiple pages with consecutive PAs can be simultaneously stored in the same cache way. If C is the number of pages that can fit in a way, then any page in physical memory can be assigned a *color* from 0 to $C - 1$. If a page with color A is cached, its lines can never evict cache lines that belong to a page with color B, as long as $B \neq A$. *This is the principle behind color-based cache partitioning that has been largely explored in the literature* [28], [25], [34].

3) Cache Replacement: The value of the index bits in the VA/PA determines the set in which a line will be cached. Since there are W ways, any of these could be selected to store a new line after a miss. Allocating a line after a miss is done in two steps. If any of the lines in the selected set is *invalid* – i.e. it does not contain cached content – then it

will be selected for allocation (line-fill). Otherwise, a *cache replacement policy* will be used to select which content should be evicted to make room for the new line-fill. The cache replacement policy is implemented in hardware and not controlled by software. There is wide consensus that deterministic replacement policies like Least-Recently-Used (LRU) or First-Come-First-Served (FIFO) are to be preferred in the context of real-time systems. *Nonetheless, increasingly more vendors implement random replacement policies to favour applications targeting average performance.*

C. DRAM Memory Structure

If a memory access results in a cache miss in LLC, then the request is forwarded to (off-chip) main memory. The vast majority of commercial platforms use DRAM to implement inexpensive, large-capacity main memory storage. DRAM storage is organized in multiple *channels*, *ranks*, *chips*, *banks*, *rows*, and *columns*. A number of works have covered this organization, and how it affects predictability in great detail [34], [44], [10]. For the purpose of this work, it is important to recall that different banks can process memory requests in parallel. At the same time, throughput optimization strategies implemented at the DRAM memory controller can typically reorder requests targeting the same bank. It follows that allocating physical memory in different banks for applications on different cores is crucial to prevent undesired bank-level contention [44].

A group of bits in the PA determines the channel, rank, chip, and bank where a given PA maps to. Clearly, a different bank is selected as long as two PAs differ in any of these bits. As such, we collectively call these bits as *bank-bits*. The exact position of bank-bits in the PA is strictly vendor-specific. Moreover, bank-bits may be arbitrarily distributed in the PA.

D. Illustrative Example

In order to have a global understanding of the interplay between addressing spaces, caches, and DRAM mapping, let us construct an illustrative example. For this example, we use parameters from one of our evaluation platforms: the NVIDIA Tegra TX1 SoC. More details about this system are provided in Section VI.

Let $L_S = 64$ bytes and a 34-bits addressing space[1]. Consider a 32 KiB 2-way set-associative PIPT L1 cache. In this case, we have 4 colors in L1, encoded by PA bits [13:12]. The L2 (and LLC) is a 2 MiB 16-way set-associative PIPT cache. As such, we have 32 colors encoded by bits [16:12]. Finally, the system has 4 GB of DRAM memory with a total of 16 banks across 2 chips. The bank bits are [31, 12:10].

Figure 2 shows how the same PA is interpreted from 4 different points of view. At the top (Figure 2, OS), the bits of a PA are seen from the perspective of an OS/Hypervisor as divided into Page Frame Number (PFN) and Page Offset (PO) bits. For the typical page size of 4 KiB, PO bits are [11:0]. Via one- or two-stage memory virtualization, it is possible to directly control only the PFN bits to which a given VA/IPA maps. Figure 2 (L1) depicts the structure of the same PA from the point of view of the L1 cache, highlighting its color bits. Similarly, Figure 2 (L2) reports the PA structure from the point of view of the L2 cache. Finally, Figure 2 (DRAM) highlights

[1]In modern 64-bit systems, it is never the case that all 64 bits are truly implemented. Typical systems use between 33 and 40 bits.

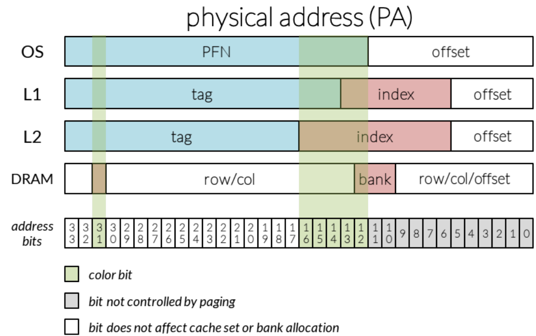


Fig. 2: Role of bits in a 34-bits PA address from the point of view of an OS/hypervisor, L1 and L2/LLC (PIPT) caches, and DRAM controller.

the bank bits used by the DRAM memory controller. It follows that the color bits for a page are bits [31, 16:12], and that the system has 64 colors in total.

IV. Coloring-Based Partitioning

By carefully performing allocation of PAs to user-space applications, it is possible to *partition* access to resources in the shared memory hierarchy. This technique is called *page coloring* and it represents a powerful tool to achieve temporal isolation between memory-intensive applications on different cores. In this section, we provide an overview of the practical challenges and the often overlooked side effects of cache/DRAM bank partitioning via coloring. For each we propose a possible solution and/or mitigation.

A. Coloring Strategy for Strict Partitioning

In this work, we focus on *strict partitioning*. Under strict partitioning, each core is assigned a set of colors and all the applications activated on a core inherit the same color assignment. Changes in colors-to-core assignment can be made, as discussed in Section IV-D, but this is considered a rare operation performed at the boundaries of major system mode changes.

While the intuition behind coloring is straightforward, its practical implementation can be tricky. For systems with one-stage address translation, an OS should be modified to ensure that applications are allocated physical memory from a different *pool* of pages depending on the core on which they will be executed. This has been successfully implemented in a number of works [34], [43], but the approach is less than ideal. In fact, intrusive and non-trivial modifications are required in the OS to maintain multiple colored page pools. Additionally, applications still cross the boundaries of their colors when interacting with the OS itself. Finally, special care must be taken for applications that share dynamically linked libraries [14].

A cleaner approach consists in leveraging virtualization extensions [28], [12]. In this case, a hypervisor and hence a two-stage translation regime for VAs is activated. The hypervisor activates separate OS's. Each OS is unmodified and sees a contiguous space of IPAs. The hypervisor however maps IPAs of different OS's to PAs with non-overlapping colors.

A major drawback of this approach is that hypervisor code is rather platform-specific, with only a limited number of supported SoCs. The majority of a hypervisor's code-base

is used to perform SoC-specific bootstrapping and virtual CPU (VCPU) scheduling. If the goal is strict partitioning and strong temporal isolation between partitions, however, VCPU scheduling is not strictly required. Instead, a static CPU-to-partition assignment is to be preferred. Similarly, booting the system into hypervisor mode is not the best approach since a traditional OS contains already all the logic to perform bootstrapping.

This paper proposes an alternative approach, called *Boot-first, Virtualize-later*: we boot the system via an unmodified OS, called the *root partition* (e.g. a standard Linux kernel). Next, if needed, we activate a hypervisor that dynamically virtualizes the root partition, activates two-stage VA translation, and provides an interface to define and activate additional partitions. Our hypervisor is also kept to the bare minimum and does not perform any VCPU scheduling. As such, we refer to it as the *partitioning hypervisor* (PH). The resulting simplification of PH's code-base enables quicker porting to newer SoCs and enhances system maintainability. In this work, we have extended Jailhouse [16]: a lightweight PH that already implements the *boot-first, virtualize-later* principle.

In the rest of the paper we use the term "partition" to refer to a specific OS being virtualized by our PH.

B. Expressive Color Representation

As discussed and illustrated in Section III-D, color bits are multiple and potentially non-contiguous in the PA. This raises the question of how to express the color selection in a way that is compact and yet general enough to capture all possible hardware configurations.

Recall that each partition needs to access a number of PA apertures. Some of these correspond to normal memory (DRAM), others correspond to memory mapped devices. When (re-)configuring a partition, the PH needs to distinguish between regular memory to be colored and device apertures that should not be colored[2].

Each partition in Jailhouse is associated a set of memory region descriptors, defining how PA memory apertures will need to be mapped to IPAs. In our extension, we added a *color specification* to each of such descriptors.

A color specification is really a *restriction* on the PAs that can be assigned when performing the IPA-to-PA mapping. We express coloring requirements in the most generic way via two additional fields in a memory region descriptor: (1) a *color-bits mask*; and (2) a *color-bits value*. Simply put, the color-bit mask defines which bits in a PA correspond to color bits. Which exact value should be enforced for those bits is then specified in the second field, the color-bit value.

Example: if all the color-bits for the setup considered in Section III-D were considered, the mask will be `0x0_8001_F000`, selecting bits [31, 16:12]. Let us assume that we have two partitions to be colored so that they occupy two different DRAM banks, do not overlap in L2, but where each can fully use its own L1. Since bit 12 and 13 would result in unwanted L1 partitioning, these need to be cleared from the mask. To partition the L2 in two parts, we can assert any one bit among bits 14, 15, or 16 in the mask. Finally, asserting bit 31 is necessary to enforce DRAM bank partitioning. A possible color-bit mask for both partitions will be `0x0_8000_8000`. In terms of color-bit values, the first partition can be assigned `0x0_0000_0000`, placing any PA mapped to this partition in the first half of L2, and on the first group of banks. The second partition can be assigned color-bit value `0x0_8000_8000`, placing it in the second group of banks and in the bottom half of L2.

[2] In fact, coloring a device aperture would not result in partitioning access to the I/O device, but only in making a portion of the device's configuration space inaccessible.

C. Efficient Selection of Color-compliant PAs

When a partition is bootstrapped, it is responsibility of the PH to map that partition's IPAs only to PAs that satisfy the color requirements.

Expressing color requirements in the generic way detailed above, however, has a major problem: there could be a large gap between two PAs that satisfy the color requirement. This gap is also not constant. *Example:* consider the color-bit mask `0x0_8000_8000` and color-bit value `0x0_0000_8000`. The PA address `0x0_7FFE_E000` clearly satisfies the requirement. The next page whose PA satisfies the requirement is at `0x0_7FFE_F000`, so only 1 page away. But the next page after that is at `0x0_7FFF_8000`, so 9 pages away. Also, once we reach `0x0_7FFF_F000`, the next useful page is `0x1_0000_8000`, so 524,297 pages away.

It follows that just linearly scanning the range of PAs looking for the next color-compliant address can be extremely inefficient. To solve this issue, we have devised an efficient algorithm to directly generate, given a PA, the closest color-compliant address. The algorithm can be used to compute the next color-compliant address in $O(1)$ and uses only a series of bit-wise operations. The full algorithm is reported in the appendix (see Listing 2).

D. Dynamic Re-Coloring

As previously mentioned, offloading the duty of booting the SoC to the OS, and then activating the PH has multiple advantages. First and foremost, it allows keeping the PH code-base at a bare minimum. Additionally, it allows supporting mode-changes from a PH-less configuration with a single SMP OS, to a multi-partition configuration, and back.

Recall that when the SoC boots without a hypervisor, the OS in the root partition has direct access to physical memory. It will map itself and all its applications into contiguous PA pages. Consider now what happens when the PH is activated. The set of PAs that the root partition has been using from boot to PH activation is not colored. Albeit not color-compliant, these pages may contain important state/data required by OS and applications to correctly execute. It follows that before creating a color-compliant IPA-to-PA mapping for the root partition, the content of these pages needs to be moved to the new set of colored PAs. *Example:* assume that the root partition used three pages with PA `0x0_0000_0000`, `0x0_0000_1000`, and `0x0_0000_2000`. Assume that the color requirement for the root partition after PH activation selects odd pages only (i.e. color-bits value = `0x0_0000_1000`). The following will be the IPA-to-PA mapping for the three pages: (1) IPA `0x0_0000_0000` $\rightarrow$ `0x0_0000_1000`; (2) IPA `0x0_0000_1000` $\rightarrow$ `0x0_0000_3000`; and (3) IPA `0x0_0002_0000` $\rightarrow$ `0x0_0000_5000`.

In general, after activation of the PH, a PA A_{old} will be remapped to a different color-compliant PA A_{new}. The following steps need to be performed. (1) Suspend execution

of the root partition; (2) compute the value of A_{new}; (3) copy data from A_{old} to A_{new}; (4) update second-stage page tables so to map IPA A_{old} to PA A_{new}; (5) flush caches and TLBs as needed. When the root partition is resumed, any attempt to access A_{old} will result in accesses to (colored) A_{new} in physical memory.

This technique goes under the name of *dynamic re-coloring*. We have successfully implemented dynamic recoloring in our extension of the Jailhouse PH. To the best of our knowledge, this is the first work to propose a working implementation of dynamic re-coloring for a running OS/partition. Albeit simple in principle, the practical implementation of dynamic re-coloring presents a number of technical challenges. We discuss two of these more in detail.

1) Efficiency: A first challenge is how to copy data for an entire partition. Consider the example above. After coloring, IPA `0x0_0000_0000` should map to PA `0x0_0000_1000`. But the content that was previously in PA `0x0_0000_1000` should be copied to `0x0_0000_3000`. In other words, if pages are greedily copied to their new colored location, one ends up overwriting the content of pages that themselves need to be copied. A naïve solution consists in first copying the entire partition in a temporary *swap space*, and then copying back all the pages in their final colored location. Apart from requiring twice the number of memory transfers, this solution is not practical as the DRAM space may not be sufficient to accommodate the required swap space.

An efficient solution consists instead in copying pages backwards. With this solution we keep the number of copy operations to their minimum and do not require an additional swap space. The only catch is that an algorithm to calculate the *previous* address of a color-compliant page is required. This can be implemented in a way that is very similar to what presented in Listing 2 and omitted due to space limitations.

Note that following a similar approach, it is possible to un-color a partition when the PH needs to be unloaded, or the system undergoes a mode change that requires a different partitioning scheme (e.g. a new partition is enabled).

2) Coherence: The second practical challenge for dynamic re-coloring – and PH-based coloring in general – is that normally CPUs are not the only entities that access physical memory. In fact, a typical OS/partition interfaces with a number of I/O devices. We discuss the implications and our solution to this problem in Section IV-F.

E. Interplay-aware Color Selection

While coloring is a powerful technique to enforce memory resource partitioning, it also presents many pitfalls. Consider once again the example in Section III-D. Here, note that bit 12 is an index bit for L2, and also a DRAM bank bit. One could then attempt a setup where two partitions map to pages with different bit 12 value, to partition memory resources in DRAM and L2. Unfortunately, however, bit 12 is also an index bit for L1. This implies that if such a configuration is selected, each partition will only use half of the already constrained L1. In this case, the performance hit is almost certainly higher than allowing free contention on L2 and DRAM banks.

Additionally, note that given the example presented in Section III-D, it would be impossible to setup 4 partitions such that: (1) each accesses a different portion of L2, (2) each is assigned a different set of DRAM banks, and (3) each uses the entire private L1 space.

A solution in this case consists in selecting a SoC where the L1 is VIPT, instead of PIPT (see Section III-B). The current trend in commercial platforms is an almost uniform coexistence of SoCs with PIPT vs. VIPT L1 caches. For instance, ARM Cortex A57 processors use PIPT L1 caches, but ARM Cortex A53 processors use VIPT L1 caches.

Last but not least, it is necessary to understand the impact of coloring on the behavior of I/O devices, as we discuss in Section IV-F.

F. Coloring and I/O Devices

A truly practical implementation of coloring should consider I/O devices. Reasoning on coloring from the point of view of the CPUs alone is sufficient only for low-performance systems with extremely simple polling-based device I/O. The current mainstream trend is towards interrupt-based I/O, where DMAs and DMA-capable devices (we will just refer to these as DMAs, for short) are largely widespread.

In a nutshell, a DMA behaves like a master on the memory bus, and thus it can initiate transfers of I/O data from/to main memory. Unfortunately, however, memory accesses originated by DMAs do not undergo VA/IPA translation at the MMU, because the MMU represents CPU-specific hardware circuitry. In other words, DMAs by default can access directly PAs, which breaks coloring.

Note however that the activity of a DMA cannot trigger allocation (and hence eviction) in a CPU cache. As such, the DMA alone cannot break L2 partitioning. Conversely, the activity of a DMA can cause interference on DRAM banks. It follows that a simple solution if DRAM bank partitioning is not required, is to locate DMA buffers is contiguous (non-colored) PAs that are marked non-cacheable. Disabling cacheability for these buffers is necessary. Otherwise, a CPU retrieving/preparing data after/before an I/O operation may cause L2 evictions outside of its assigned partition.

Simply setting DMA buffers as non-cacheable has a number of disadvantages. First, as previously mentioned, it does not prevent DMAs from violating DRAM bank partitioning. Second, it is not suitable for high-performance systems. In high-performance systems, in fact, it is very common for DMAs to be *coherent* with CPU caches. Cache-coherent DMAs provide a boost in performance for I/O intensive applications. In fact, they allow applications to perform in-place processing of fresh I/O data, i.e. without requiring additional memory copies. Clearly, setting a DMA buffer as non-cacheable does not allow exploiting this important architectural feature. Third, in modern SoCs, it is possible for DMA engines to allocate cache lines[3]. This is useful to warm-up the cache with I/O data. The mechanism cannot operate if DMA buffers are non-cacheable.

Modern SoC vendors have become aware of the problems that arise from DMAs being able to direct access PAs. As a result, modern SoCs include so called System MMU (SMMU) circuitry[4]. By leveraging SMMU support, it is possible to control the mapping between IPAs and PAs as seen by DMAs and DMA-capable devices. Hence, it is possible to restrict DMAs to access only colored PAs. Ultimately, an SMMU

[3]The technology goes under the name of Direct Cache Access (DCA) in Intel platforms [11].

[4]The term SMMU is widely used in ARM-based SoCs, but it is frequent for the same component to be called IOMMU – see Intel VT-d and AMD IOV extensions.

enables the use of cache-coherent DMAs while enforcing strict partitioning of LLC and DRAM banks.

In our extended Jailhouse-based PH, we have added support for SMMU. The SMMU is configured as follows. If a partition activated by the PH is configured to have access to a DMA-capable device, then SMMU support is enabled for the device. Additionally, a set of SMMU-specific page tables are initialized to create an IPA-to-PA mapping, for the considered device, that is consistent with the partition's IPA-to-PA mapping. It follows that as long as coloring is appropriately configured for a partition, then all the devices mapped to the same partition will only access colored physical memory.

G. Two-Stage Translation Overhead

One more important aspect needs to be considered when leveraging coloring: its side-effects on the TLB.

In a platform where the MMU has been disabled[5], an address referenced by a CPU is directly a PA. As such, no translation is required and an immediate lookup in a PIPT cache can be performed. If the MMU is enabled by the OS to setup a multi-programming environment, user-space applications will reference VAs. A VA needs to go through first-stage translation to determine the corresponding PA. Depending on the type of mapping between VAs and PAs, the lookup itself may require one, two, or three additional memory accesses to complete a page-table walk. In a PIPT cache, this has to be done before cache lookup can even be performed.

The impact of implementing coloring at a hypervisor-level is twofold. On the one hand, VAs translate at the first stage only into IPAs, that need to go through a second translation stage before the corresponding PAs can be determined and accessed. This drawback is well-known and affects any system that leverages hardware virtualization.

In order to reduce the performance hit of a two-stage translation process, a hypervisor typically maps IPAs so that large contiguous blocks of IPAs (e.g. a 1 GB aperture) map to large contiguous blocks of PAs. In this way, second-stage translation can be limited to use only one level of page tables – and thus incur only one extra memory reference for a VA translation. Unfortunately, however, the granularity required for a colored mapping rules out this possibility. In other words, implementing coloring by leveraging virtualization means that the hardware has to perform a total of seven memory accesses whenever a VA is referenced by a user-space application.

In this sense, the TLB plays a crucial role. In fact, the TLB caches translation results for VA-to-PA addresses. This way, the price for a full two-stage page table walk is only paid in case of a TLB miss. As we discuss in Section VI-B, the cost of a TLB miss can become significant and requires being appropriately bounded when timing analysis of applications is carried out.

V. Deterministic Cache Content Management

As mentioned in Section III-B, caches in modern SoCs do not support locking and often implement a random replacement policy. In this section, we present a novel technique,

[5]Using a modern SoC with MMU disabled is a terrible idea. First, it is often the case (e.g., all ARMv7 chip families and ahead) that cacheability attributes are encoded in page table descriptors. When MMU is disabled, all memory is treated as non-cacheable. Second, VIPT caches, that are common in embedded space and the norm in desktop- and server-grade CPUs, are rendered useless when virtual memory is disabled.

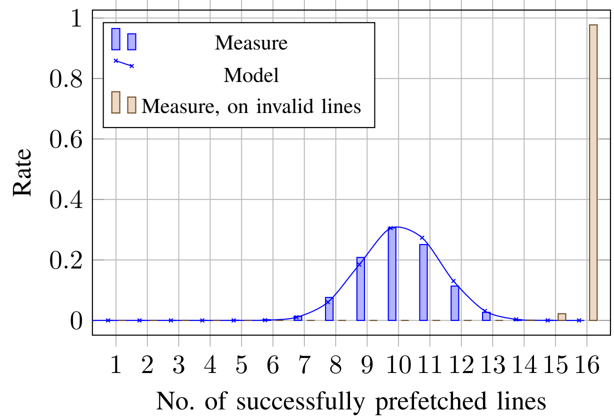


Fig. 3: Success rate and probability for 16 prefetches. Cache set lines before the prefetch are assumed to be valid and clean.

namely *invalidation-driven allocation* (IDA) to deterministically control the content of a random replacement policy for applications with small footprint. We also provide insights on how the technique can be extended to applications with larger footprint.

A. Motivation

Randomness in the cache replacement policy exposes a fill sequence of useful data to self-eviction that will later cause additional miss, and this anomaly is exacerbated by the presence of page coloring, which increases the chances of high utilisation of the same set.

We shall introduce this issue with a concrete example, referring the reader to the (involved) probabilistic analyses that can be found in the previous literature [2], [1], [17], [19], or to a brief (and gentle) introductory description in Appendix A. In an experiment detailed in Appendix B1, 16 congruent addresses are loaded in a pseudo-random-replacement 16-way cache. The blue bars in Figure 3 plot the distribution of the number of addresses that are correctly cached. The average is around 10.16, therefore accesses to 35.6% of the lines will afterwards miss the next first access. In the following, we shall introduce a technique to significantly improve this result, obtaining the performance shown in the rightmost bar, allowing to obtain 16 successfully prefetched lines at probability close to 1.

B. Basic Principle

Recall from Section III-B that in a typical CPU cache, the cache replacement policy is invoked to select a line to evict *only if* none of the lines in the selected set is invalid. Conversely, if an invalid line exists, it will be deterministically selected without evicting any other valid line. Our technique leverages this principle. More specifically, when a task begins or resumes execution, it suffices to make room in the cache by cleaning—i.e., writing back any dirty cache line to main memory—and invalidating a number of lines that are expected to be used by the task. This technique, called *invalidation-driven allocation* (IDA), ensures that new lines accessed by the task will not overwrite one with another.

C. Extensions

IDA in itself only works if applications have a memory footprint that is smaller than the assigned LLC partition. This

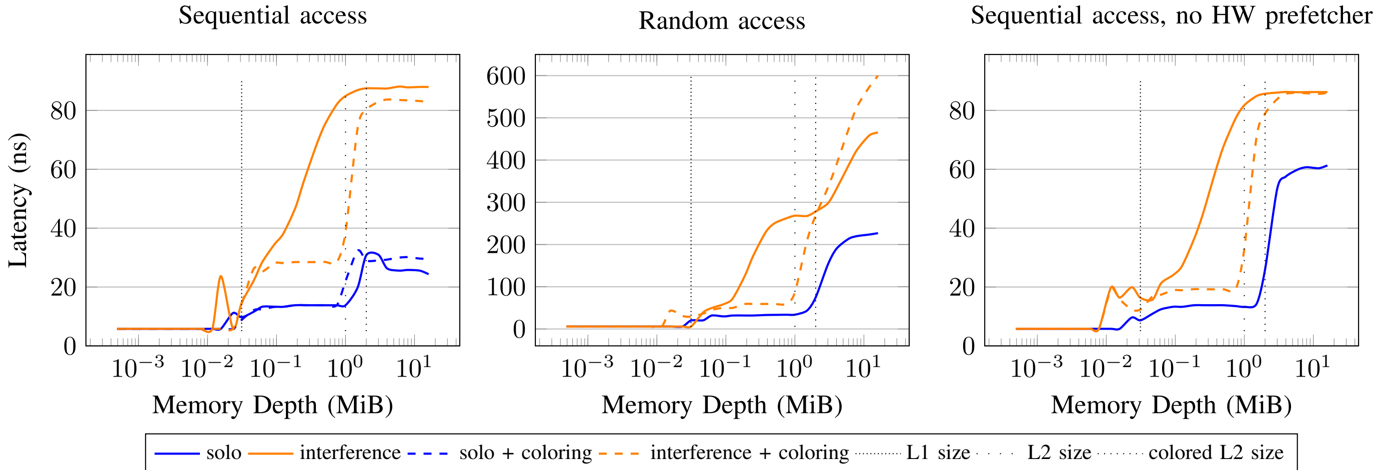


Fig. 4: Memory Latency Benchmark with Coloring.

is because once all the cache lines have been filled as the task execute, additional accesses will cause non-deterministic line replacements. There are two viable ways to extend IDA when applications with larger working set are used.

The first approach can be applied in case of applications that undergo multiple phases, each operating on a small working-set. In this case, it is possible to exploit compiler-time memory analysis to appropriately insert cache invalidation operations at the boundary of phase changes. A similar technique was successfully applied in [33] to split a task at compile time to progressively load its memory via a sequence of DMA operations.

A second approach consists in leveraging *memory types*. Modern SoCs provide additional bits in the page-table descriptors to encode how each memory page should be handled by the hardware. In this case, it is possible to selectively specify that a page of main-memory should be treated as non-cacheable memory. Given an application with large footprint, first memory profiling is used to identify those pages that provide the maximum benefit when allocated in cache. Next, no more pages than the size of the assigned cache partition are marked as cacheable, while any other page is marked as non-cacheable.

The two approaches above can also be combined to dynamically change the set of cacheable and non-cacheable pages as the application progresses. In this way, the cache content can be updated following working-set changes in the application.

VI. Evaluation

We hereby evaluate the behavior of the implemented system on a Nvidia Tegra TX1 SoC, whose main features are summarized in Table I. We focus first separately on (1) coloring, (2) virtualization, (3) IDA; and then (4) on the integrated framework.

A. Coloring-based Partitioning

We measured the memory latency for a partition within our PH, observing the impact of external memory interference, and testing a coloring setting. In particular, we used Jailhouse 0.8 with the described extensions and four partitions, using one core each, all running Linux 4.14. The *test cell* runs `lat_mem_rd` microbenchmark from LMBench 3 [27].

1) No Partitioning: Two *interfering cells* run an application heavily polluting the shared L2 cache. We employed `stress` 1.0.4 [39] which repeatedly reads and writes an 8 MiB memory area, in sequential order, with a 64 B stride distance, and without any reallocation. Measurements without cache coloring for sequential and random access are respectively plotted as solid lines in the two leftmost plots of Figure 4. Not surprisingly, when the memory footprint fits the L2 cache but exceeds L1 size (i.e., for sizes between 32 KiB and 2 MiB), the presence of the two interfering cells (“interference”) makes the memory access latency grow almost linearly with the footprint. Moreover, in both graphs we observe that latency increase is a little slower between 32 and 96 KiB. This is most probably an effect of ARMv8 cache implicit lockdown, which has been recently studied in [26]— a cache line in L2 cannot be evicted as long as it is cached in one of the L1 caches due to the inclusive relationship between the two levels.

2) Coloring: We repeated the previous experiment with a colored configuration, and assigned 16 colors to the test cell and 8 to each interference cell (i.e., 1 MiB and 512 KiB of L2 cache, respectively). These experiments are reported as dashed lines in Figure 4. The results show that coloring allows significantly decreasing the memory access time when the application’s footprint can fit in the L2 partition. The maximum speedup is 65.2%, or 77.4%, for sequential and random access, respectively. Such a speedup is no longer depth-dependent, but it stays around 26∼28 ns and 37∼59 ns for sequential and random accesses, respectively. On the other hand, there are three drawbacks which deserve to be highlighted. The first and obvious one is that the per-partition size of L2 is now limited to the assigned partition, i.e., 1 MiB. Secondly, coloring is not able to completely avoid interference on L2 cache, even if the memory footprint fits in the L2 partition. We further study this problem in the next experiment. Finally, coloring significantly increases memory access latency when the memory footprint is considerably larger than L2 and memory accesses are randomly distributed. This arises from a side-effect of virtualization-based page coloring and is further discussed in Section VI-B.

3) Coloring and Cache Interference: When the memory footprint fits in L2, we observe a non-negligible overhead (up to 4.59% increase) on memory access times when comparing a colored system with interference against an unmanaged cache

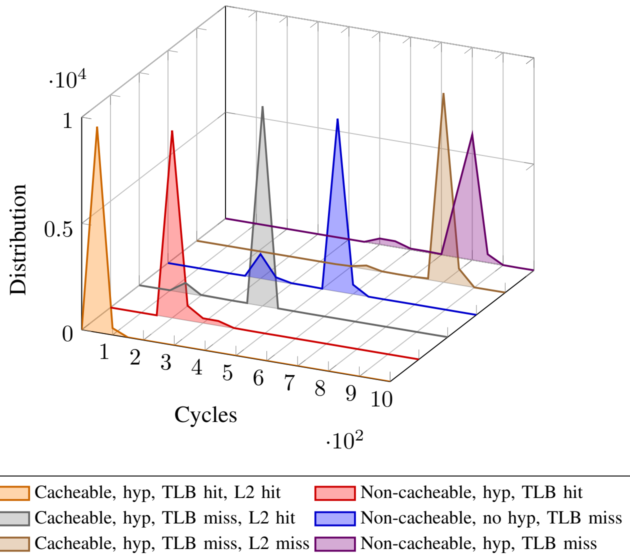
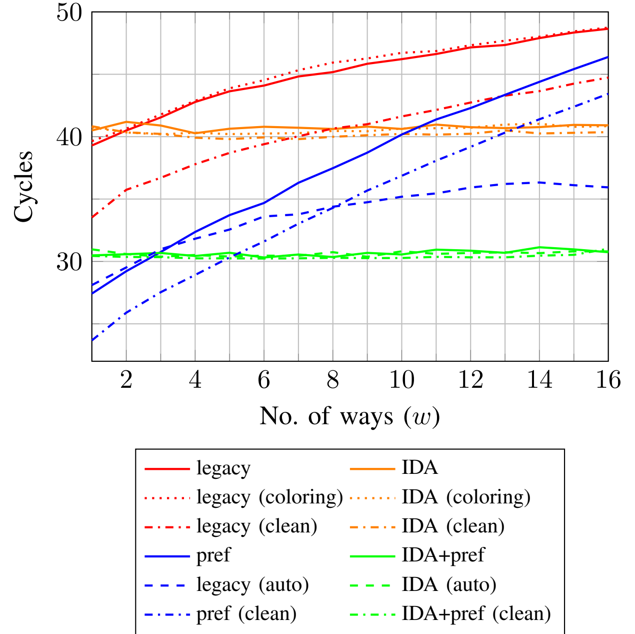
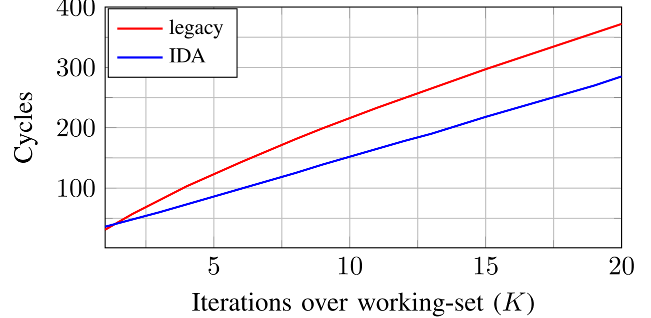


Fig. 5: Distributions of L2 and TLB access time with or without hypervisor.



(a) Per-reference memory access time, with $K = 2$ rounds over the working-set, and w number of used ways.



(b) Execution time on 16 ways, as a function of K.

Fig. 6: Microbenchmark results.

without interference. A first cause of such an effect may be ascribed to the hardware (HW) automatic prefetcher[6], which generates a number of load requests competing for a shared cache component, such as the coherent interconnect bus or the snoop control unit, that can be saturated by the benchmark alone. Another possibility is that NVIDIA implementation of the HW prefetcher, violating ARM specification to maximize performances, is allowed to cross page boundaries, hence loading useless lines across color boundaries. These interpretations are confirmed by disabling the HW automatic prefetcher and repeating the memory latency benchmark for sequential access. Results are in the righmost plot of Figure 4, where it is easy to notice that the latency drops from about 28 to 19 ns, despite the interference of two stressing inmates. We conjecture that the remaining 6∼7 ns overhead are the effect of contention on MSHRs, which hold the status of pending miss accesses, and that are saturated by the interference inmates. This effect and the measured entity of its impact is coherent with the results shown in [38], which recently investigated this problem.

B. Virtualization-based Coloring Overhead

In presence of external interference, and when coloring is enforced, considerable overhead is measured in the latency of a memory operation with random access pattern on a footprint exceeding L2's capacity. This is caused by the cost of a miss in the *translation look-aside buffer* (TLB), as corroborated by the following tests. Here, we compare the latency for memory accesses (performed with the `ldr` instruction) in the following cases: (1) a cacheable or (2) non-cacheable region is being accessed, in a system where the PH is (3) enabled or (4) disabled, where the accessed address causes (5) a TLB miss, or (6) a TLB hit. Figure 5 presents a histogram with the access time distribution. Notably, accesses to cached or non-cached data when the corresponding address translation is in the TLB is not affected by the presence of the PH. The case without hypervisor is thus omitted in Figure 5. Upon a TLB miss, instead, we observed that access latency grows from 550 to 800 cycles when the hypervisor is introduced. Indeed, when the requested VA is not present in the TLB and virtualisation is enabled, translation must undergo two stages of page table walk. Conversely, in a setup where virtualisation is disabled, only one stage of address translation is performed. The impact of TLB misses is therefore higher in a virtualized environment. Since random accesses are much more likely to incur in TLB misses, a performance degradation for large memory footprints is visible (only) in the middle plot of Figure 4.

C. Invalidation-Driven Allocation (IDA)

We evaluate the effectiveness of IDA use synthetic benchmarks, as well as real benchmarks. We hereby summarize the results of our experiments.

1) Syntetic benchmark: We designed a synthetic benchmark so that its working-set size can fit in w L2 cache ways. The parameter w can be adjusted from run to run. Given the geometry of our LLC, we consider $1 \leq w \leq 16$. The benchmark is composed of two phases: *preparation* followed by *execution*. During the preparation phase, the benchmark can perform the following: (1) "legacy" – nothing; (2) "IDA" – cleaning and invalidation of w ways; (3) "pref" – prefetch of its entire

[6] This does not imply we are (yet) advocating for deactivating automatic prefetcher—its prefetch distance is finely tunable, and the best configuration depends on a wide spectrum of application-specific parameters.

working-set; or (4) "IDA+pref" – cleaning and invalidation of w ways (IDA), and prefetch of its entire working-set. During the execution phase, the benchmark performs accesses within its working-set, accessing its entire memory K times. The time for its entire length is measured.

Additional configurations are related to the system setup. These are: (1) "auto" – the automatic hardware prefetcher is enabled; (2) "coloring" – PH-based coloring is enabled and configured so that the size of each way is reduced in half; and (3) "clean" – the cache is initially clean, i.e. without pending write-backs. The micro-benchmark is always executed inside the PH, and it is executed 20 times in each configuration.

2) Results: Figure 6a shows the number of cycles needed to run the benchmark as a function of w. All the times are normalized by the total number of performed memory accesses. The following conclusions can be made. First, in the legacy/baseline case without prefetching (red lines), the time grows with the number of used ways from around 39 to 47 cycles. Adding prefetching (blue lines) yields a visible improvement. For more than 3 ways, the automatic prefetcher performs better (28 to 36 cycles) than the "pref" case with manual prefetches (33 to 46 cycles).

A key result is that with IDA alone (orange lines), instead, access times remain near-constant and at around 41 cycles. Hence, IDA outperforms the pure legacy cases for $w \geq 3$ if dirty cache lines exist, and for $w > 7$ for a clean cache.

A second key takeaway is that by combining prefetching and IDA, the access time further drops down to roughly 31 cycles/line. In case of both automatic and manual prefetching. If compared with the baseline, we have a reduction on the average execution time by more than 20% with $w = 1$, and up to 30% with all $w = 16$. Finally, note that enabling cache coloring does not negatively affect the performance of IDA.

Figure 6b shows the results when $w = 16$ and the number K of iterations over the working-set in increased, comparing IDA with the legacy baseline. Under IDA, the execution time slope (blue line) is constant near 13 cycles/iteration, meaning that the latency introduced by any additional line is determined only by the L2 cache access time. In the legacy case, the slope of the (red) curve is around 25 cycles/iteration during the initial first rounds, because due to the random replacement policy previously allocated lines may have been evicted before the next iteration (self-eviction). The slope gradually settles around 15 cycles/round, but the overall execution time remains higher. Note that if the WCET was derived via static analysis the gap between the curves would be much wider.

D. Integrated Framework Approach

To validate the whole PH solution we are proposing, we measured the performance and predictability gain in realistic scenarios. We ran commonly used algorithms within one or more cells of our virtualized solution with cache coloring.

1) Description: We consider a mix of application from the Linux kernel library and the TACLeBench benchmark suite.

Each applications was executed 1K times inside a PH-enforced partition running on a single core. The partition assigned to each benchmark when coloring is 64 KiB (1 color). This size is in line with their working-set size. The cache is always initialized as dirty, hardware prefetch and branch prediction are disabled. Beside collecting the baseline configuration described so far, we test the following configurations: memory interference is induced by other cores (see Section VI-A) without ("Interf.") and with ("Interf.+Color") coloring; and using IDA on the benchmark partition without ("IDA") and with ("IDA+Color") coloring.

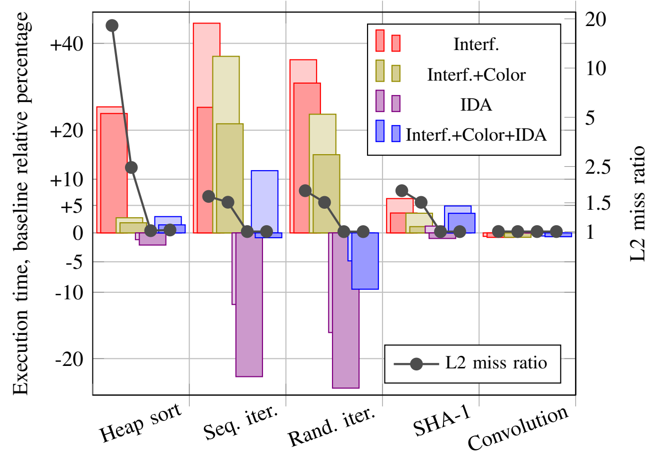


Fig. 7: Real benchmark results: average and maximum execution time (strong vs faint tint, respectively) relative to the baseline average value; and L2-cache miss ratio.

2) Results: We summarize the results obtained with real benchmarks in Table II. A few remarks follow. First, coloring or IDA brings no appreciable difference to SHA-1 benchmark, nor Convolution—the applications' working-set easily fits in the L2 partition and/or the L1 cache, thus hiding any effect of the presence/absence of non-determinism and contention at L2/DRAM. Thirdly, IDA without interference improves the average and worst-case execution time compared to the baseline. The improvement is particularly significant for the "Seq. iter" and "Rand. iter" benchmarks. Furthermore, when interference is enabled, in 2 out of 5 cases, coloring alone is not sufficient to solve the problem of contention. Conversely, combining IDA with coloring yields comparable results to (first, fourth, and fifth benchmark), or outperforms coloring alone (second and third benchmarks). Indeed, by looking at the number of L2 cache misses, it can also be noted that IDA is always able to keep the number of L2 cache miss per line very near to the theoretical value of 1.

VII. Conclusion

In this work, we conducted an analytical and experimental study of the ability to achieve tight real-time guarantees in modern families of commercial multi-core embedded platforms. First, we proposed an implementation of a hypervisor-level memory coloring. Next, we investigated the impact of a set of traditionally neglected hardware features that have a significant impact on predictability. These include: hardware prefetchers, two-stage address translation, and self-eviction due to pseudo-random replacement policy. Finally, we introduced a novel strategy for deterministic cache allocation, namely (IDA).

References

[1] Sebastian Altmeyer, Liliana Cucu-Grosjean, and Robert I. Davis. Static probabilistic timing analysis for real-time systems using random replacement caches. *Real-Time Systems*, 51(1):77–123, 2015.

- [2] Sebastian Altmeyer and Robert I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–6, 2014.
- [3] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez, and Víctor Viñals. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *J. Syst. Archit.*, 57:695–706, August 2011.
- [4] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's first 64-bit arm processor. *IEEE Micro*, 35(2):46–55, Mar 2015.
- [5] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33:67–74, December 2000.
- [6] Heiko Falk and Helena Kotthaus. WCET-driven cache-aware code positioning. In *Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems*, CASES '11, pages 145–154, New York, NY, USA, 2011. ACM.
- [7] Farzad Farshchi, Prathap Kumar Valsan, Renato Mancuso, and Heechul Yun. The deterministic memory abstraction and supporting cache architecture for multicore real-time systems. In *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
- [8] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Comput. Surv.*, 48(2):32:1–32:36, November 2015.
- [9] Giovani Gracioli and Antonio Augusto Frohlich. An experimental evaluation of the cache partitioning impact on multicore real-time schedulers. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on*, pages 72–81. IEEE, 2013.
- [10] D. Guo and R. Pellizzoni. A requests bundling dram controller for mixed-criticality systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 247–258, April 2017.
- [11] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network i/o. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, June 2005.
- [12] H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, Oct 2016.
- [13] Hyoseung Kim and Ragunathan (Raj) Rajkumar. Real-time cache management for multi-core virtualization. pages 1–10. ACM Press, 2016.
- [14] N. Kim, M. Chisholm, N. Otterness, J. H. Anderson, and F. D. Smith. Allowing shared libraries while supporting hardware isolation in multicore real-time systems. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 223–234, April 2017.
- [15] N. Kim, B. C. Ward, M. Chisholm, C. Y. Fu, J. H. Anderson, and F. D. Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.
- [16] Jan Kiszka, Valentine Sinitsyn, Henning Schild, and contributors. Jailhouse hypervisor. Siemens AG on GitHub, https://github.com/siemens/jailhouse, 2018. Accessed: 2018-03-31.
- [17] B. Lesage, D. Griffin, S. Altmeyer, and R. I. Davis. Static probabilistic timing analysis for multi-path programs. In *2015 IEEE Real-Time Systems Symposium*, pages 361–372, Dec 2015.
- [18] B. Lesage, I. Puaut, and A. Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 171–180, New York, NY, USA, 2012. ACM.
- [19] Benjamin Lesage, David Griffin, Sebastian Altmeyer, Liliana Cucu-Grosjean, and Robert I. Davis. On the analysis of random replacement caches using static probabilistic timing methods for multi-path programs. *Real-Time Systems*, 54(2):307–388, Apr 2018.
- [20] Jochen Liedtke, Hermann Haertig, and Michael Hohmuth. Os-controlled cache predictability for real-time systems. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, RTAS '97, pages 213–, Washington, DC, USA, 1997. IEEE Computer Society.
- [21] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 367 –378, feb. 2008.
- [22] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Enabling software management for multicore caches with a lightweight hardware support. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 14:1–14:12, New York, NY, USA, 2009. ACM.
- [23] Elena Lucherini. Design and Implementation of a Memory Allocator to Achieve Cache Partitioning in the Linux Kernel. Master's thesis, Scuola Superiore Sant'Anna of Pisa, Italy, 2017.
- [24] R. Mancuso, R. Pellizzoni, N. Tokcan, and M. Caccamo. WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment. In Marko Bertogna, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:23, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [25] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 45–54. IEEE, 2013.
- [26] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In USENIX Association, editor, *Proceedings of the Second Workshop on Real, Large Distributed Systems: December 13, 2005, San Francisco, CA, USA*.
- [27] Larry McVoy and Carl Staelin. Lmbench – tools for performance analysis. http://www.bitmover.com/lmbench, 2009. Accessed: 2018-03-31.
- [28] Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms. In *Proceedings of the IEEE International Conference on Industrial Technology (ICIT 2018), Lyon, France*, Feb 2018.
- [29] Frank Mueller. Compiler support for software-based cache partitioning. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, &Amp; Tools for Real-time Systems*, LCTES '95, pages 125–133, New York, NY, USA, 1995. ACM.
- [30] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30:125–133, November 1995.
- [31] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, RTSS '02, pages 114–, Washington, DC, USA, 2002. IEEE Computer Society.
- [32] M. Caccamo R. Mancuso, R. Dudko. Light-prem: Automated software refactoring for predictable execution on cots embedded systems. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–10, Aug 2014.
- [33] Muhammad Refaat Soliman and Rodolfo Pellizzoni. WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.
- [34] N. Suzuki, H. Kim, D. d. Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692, Dec 2013.
- [35] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared L2 caches on multicore systems in software. In *Workshop on the Interaction between Operating System and Computer Architecture*, June 2007.
- [36] E. Tamura and Javeriana Cali. Towards predictable, high-performance memory hierarchies in fixed-priority preemptive multitasking real-time systems. *15th International Conference on Real-Time and Network Systems (RTNS'07)*, pages 75–84, Mar 2007.
- [37] G. Taylor, P. Davies, and M. Farmwald. The tlb slice-a low-cost high-speed address translation mechanism. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 355–363, May 1990.
- [38] P. K. Valsan, H. Yun, and F. Farshchi. Taming Non-Blocking Caches to Improve Isolation in Multicore Real-Time Systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, April 2016.

[39] Amos Waterland. stress. http://people.seas.harvard.edu/~apw/stress/, 2010. Accessed: 2018-03-31.

[40] R. West, Y. Li, E. Missimer, and M. Danish. A virtualized separation kernel for mixed-criticality systems. *ACM Trans. Comput. Syst.*, 34(3):8:1–8:41, June 2016.

[41] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. vCAT: Dynamic Cache Management Using CAT Virtualization. pages 211–222. IEEE, April 2017.

[42] J. Yan and W. Zhang. Time-predictable multicore cache architectures. In *2011 3rd International Conference on Computer Research and Development*, volume 3, pages 1–5, March 2011.

[43] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, Aug 2014.

[44] H. Yun, R. Mancuso, Z.P. Wu, and R. Pellizzoni. PALLOC: DRAM bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

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

[46] Zhenkai Zhang, Zhishan Guo, and Xenofon Koutsoukos. Handling write backs in multi-level cache analysis for wcet estimation. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, RTNS '17, pages 208–217, New York, NY, USA, 2017. ACM.

[47] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.

Appendix

A. Pseudo-Random Replacement Analysis

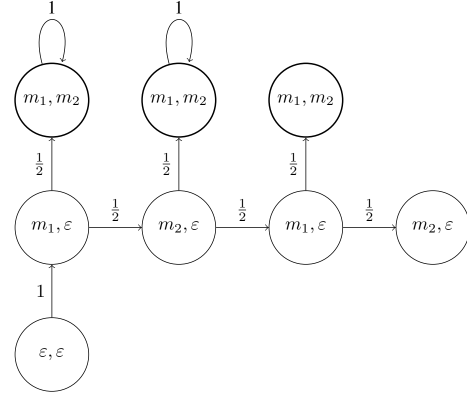
1) A qualitative description of 2-way cache: Let us consider a 2-way associative cache, and to run a number of rounds of identical access sequences of length 2 to a single given set. During each round, every memory location is accessed exactly once and always in the same order. The cache is assumed to be initialized with valid cache lines outside the given set. More crucially, we enforce that: (i) all memory blocks are congruent; and (ii) there is no other parallel process using the memory locations congruent with the given set, i.e., we are in a cache-colored setting. These assumptions imply that any cache miss other than the first one is caused by a *self-eviction*.

For example, let m_1 and m_2 be the two memory location of our choice, and let the access sequence be given by $(m_1, m_2, m_1, m_2, \ldots)$. The first access to m_1 is a cache miss: the data m_1 is copied into the cache by evicting one of the randomly chosen cache lines. The first access to m_2 is also a cache miss: the data m_2 is copied into the cache by evicting one of the randomly chosen cache lines, including the cache line holding m_1. The probability of evicting m_1 from a cache set holding W lines is equal to $1/2$. The probability of evicting any other cache line other than the one holding m_1 is $(2-1)/2$.

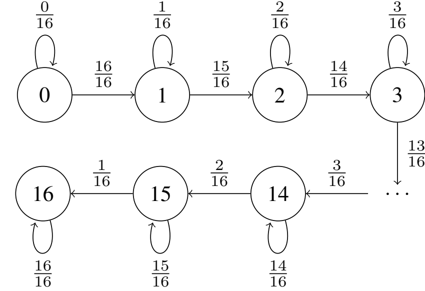
If m_1 was not evicted by m_2, the first access to m_1 during the second round is a hit. Otherwise, if m_1 was evicted by m_2, then the first access to m_1 during the second round must be a cache miss. When m_1 is copied into the cache it may evict m_2 with a probability of $1/2$. If so, the subsequent access to m_2 will be a cache miss.

Figure 8a shows the transition graph representing the possible states of the cache set over all two rounds. The symbol ε represents a cache line that does not belong to the given set of memory location. The nodes represent the cache set contents. The probability of changing to another cache set content is represented on the edge between the respective nodes. By traversing the graph starting from the initial state $(\varepsilon, \varepsilon)$, we can obtain the probability of having a given number of cache misses.

2) A quantitative description of 16-way cache: We modelled the number of useful cache locations in a 16-way cache set under a sequence of fill operations with a Markov process implemented in the GNU Octave code in Listing 1 and described in Figure 8b. The model has been validated against the ARM prefetch instruction (`prfm`), as illustrated in Appendix B.



(a) 2-way cache, states are identified by cache content.



(b) 16-way cache, states are identified by the number of useful lines.

Fig. 8: Markov chains modelling cache content over a sequence of repetitive memory accesses.

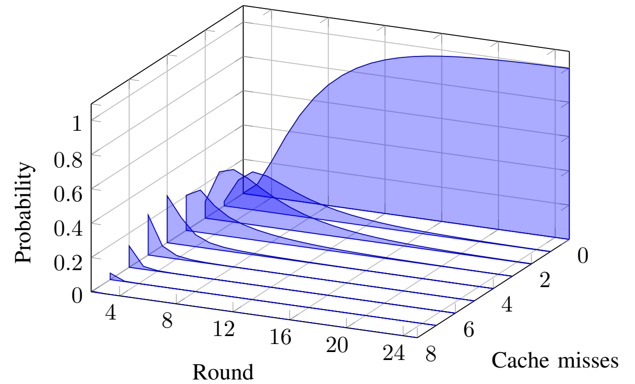


Fig. 9: Probabilistic distribution of the number of cache miss as a function of the number of accesses.

B. Pseudo-Random Replacement Experiments

To empirically verify the impact of the self-eviction pathology, we provide a set of experiments with different software prefetch patterns. To do that, we make use of ARM "prefetch memory" `prfm` instruction, which prefetches data at a given address in a given cache level.

1) Prefetch for a Whole Set: In a first experiment, repeated 13K times, we prefetch 16 congruent addresses after setting up the cache so that: (i) all ways of the set are valid, and (ii) the cache set does not contain useful data. The distribution of the number of successfully prefetched addresses is shown by blue vertical bars in Figure 3. These experimental results confirm the theoretical model given with the Markovian process (cf. Figure 8b), whose predicted performance is plotted as a solid line that perfectly matches the top of the vertical bars. In particular, the average of 10.16 successfully prefetched lines measured by the experiment matches the model's expected value of 10.303 lines. Note that the success percentage for a 16-way prefetch is only 64.393%. In the next section, we will show a technique to significantly improve this result, obtaining the performance shown in the rightmost bar with 16 successfully prefetched lines.

2) Prefetch Success for Each Way: A second experiment was designed as a variation of the aforementioned one, varying the number n of congruent lines prefetched in the considered sequence, with $1 \leq n \leq 16$. In Figure 10, we then show (i) the number of lines successfully prefetched (red histogram), and (ii) the probability of successfully prefetch the whole

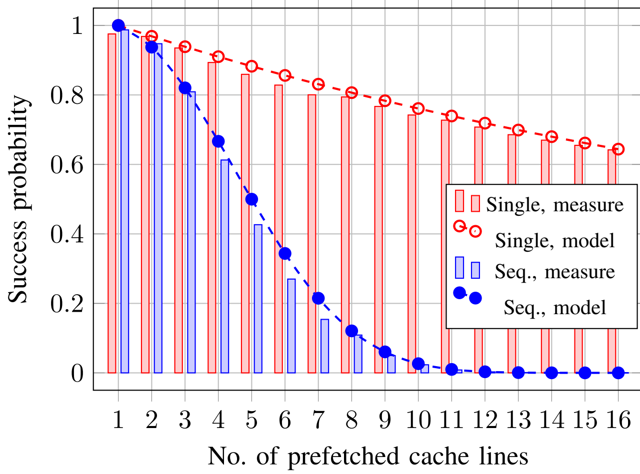


Fig. 10: Measured incidence and modelled probability for (i) the number of successfully prefetched locations (single, in red), and (ii) for successfully prefetching a complete sequence (seq., in blue), after a prefetch sequence of length n. Note that both probabilistic models are discrete. Dashed connecting lines are provided as a visual aid.

sequence (blue histogram). The experiment was performed 100K times for each number of lines to measure the former, 1M for the latter. Figure 10 shows the success rate of both events as vertical bars as function of n, while solid lines represent the corresponding probabilistic expectation, obtained analytically in the previous sub-section. We readily see that measurements are once again tightly approximated by the model. We also highlight that, while prefetching a single line has been always successful in all tries, a single occurrence of an entirely successful 16-line prefetch was encountered over 1M tries.

C. Algorithms

```
pkg load queueing

% probability that at step 0 the system is in state
p_0 = zeros(1,17);
% start always in the 0 state
p_0(1)=1;
% loop probability (staying in the same state)
p_l = 0;
% next probability (moving to the next state)
p_n = 1;
% transition probability matrix
P=zeros(17,17);

for i=1:16
    P(i,i)   = p_l;
    P(i,i+1) = p_n;
    p_l      = p_l + 1/16;
    p_n      = p_n - 1/16;
endfor
P(17,17)=p_l;
v=0:16;

% p(n,i) is the steady-state probability that the
% system is in state i after n steps.
p=zeros(16,17);
% expected values
e=zeros(16,1);
for n=0:16
    % steady-state probabilities after n steps
    p(n+1,:) = dtmc(P,n,p_0);
    % expected value
    e(n+1)   = sum(p(n+1,:).*v);
    if n>0
        s(n+1) = e(n+1)/n;
    endif
endfor
```

Listing 1: Markov chain model of L2 cache random replacement policy (GNU Octave code)

TABLE II: Real benchmarks results, data summary. Worst and average execution time, and L2 miss count per line over 1K runs.

Application	*Conf.*	*Time (Kcycles)* *maximum*	*average*	*L2 miss per line*
Heap sort	base	8,089.871	7,692.389	2.1973
	IDA	7,600.765	7,528.418	1.0127
	interf.	9,621.041	9,508.141	18.2070
	col.+interf.	7,901.555	7,831.802	2,4658
	IDA+col.+interf.	7,918.259	7,714.617	7,803.073
Seq. iterator	base	99.344	91.393	1.5107
	IDA	80.492	70.820	1.0000
	interf.	132.608	114.209	1.6406
	col.+interf.	125.038	110.916	1.5117
	IDA+col.+interf.	102.062	90.613	1.0000
Rand. iterator	base	272.061	255.475	1.5078
	IDA	214.077	193.972	0.9990
	interf.	347.411	333.316	1.7783
	col.+interf.	315.378	293.524	1.5088
	IDA+col.+interf.	243.104	231.182	0.9990
SHA-1	base	808.105	791.198	1.0000
	IDA	800.823	783.435	1.0000
	interf.	840.917	819.650	1,0010
	col.+interf.	819.320	800.049	1.0000
	IDA+col.+interf.	829.968	819.233	1.0000
Convolution	base	14,215.224	14,203.511	1.0010
	IDA	14,236.808	14,209.904	1.0000
	interf.	14,115.988	14,091.039	1.0000
	col.+interf.	14,130.744	14,092.864	1.0000
	IDA+col.+interf.	14,126.034	14,107.558	1.0000

```
ptr_t next_colored(ptr_t PA, ptr_t col_mask, ptr_t col_val)
{
  ptr_t retval = PA;
  unsigned int pos = 0;
  /* Make sure the mask value has only */
  /* bits in agreement with the mask */
  col_val &= col_mask;
  while (pos < bit_width) {
    ptr_t cur_bit_mask = 1 << pos;

    /* Skip dont-care bits, or bits that are */
    /* already compliant with the color */
    if ((col_mask & cur_bit_mask) == 0 ||
    ((col_val ^ retval) & cur_bit_mask) == 0);

    else {
      /* Clear all the bits lower than the current one */
      ptr_t reset_mask = (cur_bit_mask - 1);
      retval &= ~reset_mask;
      /* Then manipulate high bit */
      retval += cur_bit_mask;
      /* And apply lower part of color value bits */
      retval |= reset_mask & col_val;
    }
    ++pos;
  }
  return retval;
}
```

Listing 2: Find next color-compliant PA (C code).

In Listing 2, `ptr_t` is an integer type with bit-width at least as long as that of a pointer in the considered platform, and where `bit_width` is the number of bits in a PA (e.g. 34).