The Potential of Programmable Logic in the Middle: Cache Bleaching

Shahin Roozkhosh
Boston University, USA
shahin@bu.edu

Renato Mancuso
Boston University, USA
rmancuso@bu.edu

***Abstract*—Consolidating hard real-time systems onto modern multi-core Systems-on-Chip (SoC) is an open challenge. The extensive sharing of hardware resources at the memory hierarchy raises important unpredictability concerns. The problem is exacerbated as more computationally demanding workload is expected to be handled with real-time guarantees in next-generation Cyber-Physical Systems (CPS). A large body of works has approached the problem by proposing novel hardware re-designs, and by proposing software-only solutions to mitigate performance interference.**

Strong from the observation that unpredictability arises from a lack of fine-grained control over the behavior of shared hardware components, we outline a promising new resource management approach. We demonstrate that it is possible to introduce Programmable Logic In-the-Middle (PLIM) between a traditional multi-core processor and main memory. This provides the unique capability of manipulating individual memory transactions. We propose a proof-of-concept system implementation of PLIM modules on a commercial multi-core SoC. The PLIM approach is then leveraged to solve long-standing issues with cache coloring. Thanks to PLIM, colored sparse addresses can be re-compacted in main memory. This is the base principle behind the technique we call Cache Bleaching. We evaluate our design on real applications and propose hypervisor-level adaptations to showcase the potential of the PLIM approach.

I. Introduction

The modern world and society is supported by safety-critical Cyber-Physical Systems (CPS) with strict reliability and confidentiality requirements: avionic systems, nuclear plant supervisory systems, orbit controllers in satellites, space station life support apparatuses, unmanned aerial vehicles (UAVs) flight controllers, surgeon-machine interfaces in robotic surgery devices, and driverless cars, to name a few. In the last decade, we have entered a new era where bold new expectations are set on the capabilities of a CPS. Following the push for smarter CPS's, embedded platforms have substantially increased in complexity. Multi-core heterogeneous embedded systems have become mainstream for high-performance CPS's. But complexity comes at the cost of a fundamental lack of predictability. As a single backbone of memory resources is shared by multiple processors, the unmanaged contention introduces non-negligible temporal coupling between logically unrelated applications on the same SoC. The design of (1) new multi-core hardware platforms, and of (2) OS-level techniques to regulate contention have been the two main directions followed by this community to achieve performance isolation.

In this work, we outline a third possible direction. We consider modern, commercially available SoCs that include programmable logic (PL) and a traditional multi-core processor on the same chip. With this, we show that it is possible to instantiate logic that acts as an intermediary between embedded processors and main memory. By sitting between CPUs and main memory, we achieve a new degree of control over memory transactions. We call this approach *Programmable Logic In-the-Middle* (PLIM). The PLIM approach opens up new avenues to design and deploy resource management primitives previously available only though hardware re-design. It also creates new opportunities in the definition of OS-level and hypervisor-level technologies that leverage PLIM modules to improve performance management, self-awareness, and online workload characterization.

In the rest of this paper, we first lay the fundamental principles for the definition of PLIM modules. Then, we demonstrate that PLIM can be used to re-think known management primitives and solve some of their long-standing shortcomings. More specifically, we consider the problem of shared cache partitioning via page coloring. Traditional page coloring forces the allocation of sparse main memory regions. Additionally, re-coloring of applications at runtime cannot be readily performed without incurring large overheads. Thanks to PLIM, these limitations can be lifted. In summary, this paper makes the following contributions.

1) A novel approach, namely Programmable Logic In-the-Middle (PLIM), to aid the consolidation of hard real-time systems onto commercial multi-core SoCs that leverages PL to enact shared resource management;
2) We demonstrate that under PLIM it is possible to obtain an unprecedented level of inspection on the behavior of last-level caches (LLC) and main memory;
3) We show that for real applications the overhead introduced by the PLIM approach is not dramatic, justifying its applicability for use in production;
4) We solve long standing shortcomings of page coloring. Using PLIM one can prevent address fragmentation in main memory and perform zero-copy re-coloring of live applications or entire virtual machines (VM);
5) We implement and evaluate a full-stack design that includes hardware modules and hypervisor-level adaptations to take advantage of the newly available control over LLC partitioning at runtime;

II. Overview

In this section, we provide a high-level overview of the PLIM approach and provide the first key insight on how it

can be used to perform flexible cache partition management.

A. The PLIM Approach on Commercial Platforms

This work considers commercially available SoCs that integrate a traditional embedded multi-core processor system (PS) and a block of programmable logic (PL) with high-performance PS-PL communication interfaces (HPI). The PS includes full-speed processing cores, private and shared caches, a main memory (DRAM) controller, and I/O peripherals. Contrarily to SoCs built entirely on programmable logic — i.e., FPGA-only systems — the PS provides realistic performance and makes the SoC suitable to be directly embedded in production systems.

The premise of the proposed PLIM approach is that the main purpose of the PL is *not* to execute application workload, e.g., via accelerators. Rather it is used as an intermediary in the memory traffic that flows between processors and main memory. In this paper, we specifically focus on this role of the PL as intermediary logic, albeit our design does not preclude integrating traditional accelerators in the PL.

The key advantage of the PLIM approach is twofold. First, PLIM enables a level of control over memory traffic that is unprecedented for COTS systems. Second, it provides extreme flexibility because (1) when needed at run-time, it is possible to revert application traffic directly towards main memory, i.e. bypassing the PL as we demonstrate in Section VI-E; and (2) because thanks to partial dynamic re-programmable (DPR) logic, the exact control logic in the PL can be replaced at run-time. All in all, it is possible to exploit the benefits of PLIM when needed without having to commit to it for the entire lifespan of the system.

The basic mechanism that enables the PLIM approach is the ability to intercept memory transactions originated from the processors inside the PS, at the PL. In the PL, it is possible to manipulate meta-data (e.g., source/destination addresses, QoS bits), data (e.g., the payload of read/write operations), and timing of each individual transaction. Transactions are then forwarded from the PL again towards the memory controller inside the PS. In this way, the content being accessed resides in main memory, but it is possible to act on the characteristics of the traffic that now traverses the PL. We refer to this basic mechanism as **Memory Loop-Back** and discuss it in more detail in Section III. The bottom line is that with the Memory Loop-Back in place, it is now possible to add logic to enact fine-grained memory profiling and management. It is natural to wonder about the following questions. (1) *What is the overhead for transactions that are routed through the Memory Loop-Back as opposed to reaching main memory directly?* We provide a study of the overhead introduced by the loop-back in Section VI-B. (2) *What type of management can be enacted with PLIM?* We propose, implement and evaluate a first technique, namely **Cache Bleaching**, that leverages PLIM to solve long-standing shortcomings of page-coloring based cache partitioning. We briefly detail the intuition behind Cache Bleaching below and provide a more in-depth description Section IV. (3) *How can memory traffic be dynamically routed via/away from a PLIM module?* In Section V, we propose a hypervisor-level design and implementation capable of selectively and dynamically re-routing memory traffic of entire virtual machines (VMs).

B. Key Insights on Cache Bleaching

As a first case study on the PLIM approach, we consider the problem of LLC space contention between co-running application in a multi-core SoC. Cache partitioning has been deemed as an effective solution towards mitigating LLC contention [1], [2]. A typical software-only approach to perform cache partitioning is cache coloring [3]. Cache coloring can be used to partition any physically-indexed cache. This is done by carefully assigning physical memory to applications — or to entire VMs — that only maps to a subset of the available cache sets[1]. Despite being one of the core techniques to enforce performance isolation in hard real-time systems, cache coloring comes with a number of important shortcomings. Notably, it ties the amount of cache allocated to an application to the quota of the main memory that can be allocated to the same application. Moreover, the coloring selection is static because a change at run-time would require massive data movements, resulting in prohibitive overheads.

By leveraging PLIM, we can essentially redefine the meaning of physical addresses. In other words, physical addresses, as seen by the LLC, are different from those used to access content in main memory. Thus, we are able to quickly (re-)define colored physical mappings to perform LLC partitioning that results in the same contiguous main memory content being accessed. The result is the ability to perform **zero-copy re-coloring** at run-time. Because main processors see colored memory, but the color information is *stripped away* on the way to main memory, we call this technique as *Cache Bleaching*. We detail a PLIM module called *Bleacher* that implements Cache Bleaching in Section IV.

III. PLIM Basics - Memory Loop-Back

We hereby discuss the underlying mechanism that enables PLIM, i.e. the Memory Loop-Back. We provide the necessary background, some insights on the design of the module, and draw some conclusions about the overhead of PLIM on real benchmarks.

A. Background on PS-PL SoCs

A few key concepts need to be introduced to understand how it is possible to implement a Memory Loop-Back module. The PLIM approach is feasible on commercial platforms comprised of embedded non-configurable processing cores and main memory, the PS; of a block of on-chip programmable logic, the PL; and with high-performance bidirectional interfaces between the PS and the PL, referred to HPI in this work. Moreover, communication between SoC resources — e.g. between the cache controller and main memory controller — is carried out using some on-chip communication protocol. The

[1]For the reader not familiar with cache organization and coloring, we provide the necessary background in Section IV-A

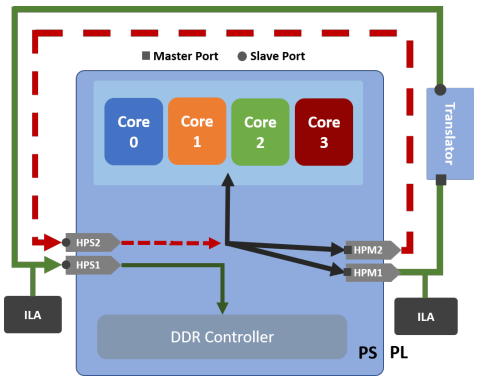

Fig. 1. Loop-back through PL. Green line with the Translator in the middle is a valid memory loop-back. Dashed red line results in an endless loop. Two logic analyzer probes (ILA) are attached to monitor the bus segments.

same protocol is also used on the HPIs. For communication to occur, an interconnected pair of ports is required. The port on which communication transactions are initiated is called the *master port*; the port from which transaction responses are sent is called the *slave port*. An SoC will have a fixed number of HPIs, each that can act as a master only, or as a slave only. While the HPIs can only be used but not instantiated, it is possible to instantiate any soft component in the PL that has master/slave ports that can be connected to HPIs. An overview with the basic blocks of a PS-PL SoC is provided in Figure 1. A survey of the commercially available families of PS-PL SoCs is provided in Table I. In the table, the PL size is reported in terms of maximum and minimum number of Logic Elements (LEs), Block RAM bits (BRAM), and DSP Slices (DSP).

To make the description of our system easier to follow, we briefly introduce the characteristics of the SoC we used. A full system implementation was performed using a Xilinx Zynq UltraScale+ MPSoC platform [4]. We use the Xilinx ZCU102 Development Board, which embeds a Zynq UltraScale+ XCZU9EG SoC. The PS is comprised of a quad-core ARM Cortex-A53 [5] cluster clocked at 1.5 GHz. Each core has a private split L1 cache (32KB instructions + 32KB data) and a shared 1 MB L2 cache, which is also the last-level cache for the A53 cores. The PS also includes a dual-core ARM Cortex-R5 cluster, which is not used in this work. Importantly, the PS also includes a main memory controller connected to a 4 GB block DDR4 DRAM. The PL is comprised of a dynamically partially re-programmable FPGA fabric with 600,000 logic cells, 32.1 Mb of memory, and 2520 DSP slices. There exist three HPIs mastered by the PS, called `HPM1`, `HPM2`, and `HPM3`. There are five HPI interfaces to initiate transactions from the PL into the PS, hence called slave HPIs. We will consider only two of them, namely `HPS1` and `HPS2`.

All the on-chip communication between functional blocks is carried out using the Advanced eXtensible Interface (AXI) protocol [6]. AXI supports two types of transactions, namely, read and write transactions. Address information and payloads for read and write transactions, and acknowledgements for write transactions are each carried via a dedicated set of signals, called channels. As such the standard defines 5 channels: (1) **AR** and (2) **AW** channels to transmit addresses for read and write operations, respectively; (3) **R** and (4) **W** channels for payload of read and write transactions, respectively; and (5) **B** channel for acknowledgements of successful write transactions.

TABLE I
COMMERCIAL PS-PL PLATFORMS

SoC	Processing System (PL) CPUs	L1 / L2 Cache	Freq.	Programmable Logic (PL) LEs	BRAM	DSP
Xilinx US+ CG	2x Cortex-A53 2x Cortex-R5	32+32KB I+D / 1 MB	1.3GHz	103K 600K	5.3Mb 32.1Mb	240 2520
Xilinx US+ EG	4x Cortex-A53 2x Cortex-R5	32+32KB I+D / 1 MB	1.5GHz	103K 1143K	5.3Mb 34.6Mb	240 1968
Xilinx US+ EV	4x Cortex-A53 2x Cortex-R5	32+32KB I+D / 1 MB	1.5GHz	192K 504K	4.5Mb 11Mb	728 1728
Intel Stratix 10	4x Cortex-A53	32+32KB I+D / 1 MB	1.5GHz	378K 10200K	30Mb 253Mb	648 5011
Xilinx Zynq-7000	2x Cortex-A9	32+32KB I+D / 1 MB	~1GHz	23K 444K	1.8Mb 26.5Mb	66 2020
Xilinx Versal	2x Cortex-A72 2x Cortex-R5	48+32KB I+D / 1MB	2.5GHz	336K 2233K	5Mb 70Mb	472 2672

Each on-chip resource that can be accessed as a slave, be it a memory-mapped I/O device, real memory, or an HPI port is assigned a range of addresses under which it responds. For non-configurable components, there is a fixed memory map with non-overlapping ranges of addresses under which each resource can be accessed. Conversely, the address under which PL components respond is programmable. For instance, when a master (e.g. a processor) performs a read transaction, the AXI master port will first use the AR channel with the address of the resource to access, and a unique ID for the transaction in flight. If the address is valid, the target resource will produce a response via the R channel. Because requests/responses are asynchronous, the produced response will have to carry the same ID used by the request.

B. Memory Loop-Back Design

What is required to implement a memory route that leaves the PS, enters the PL and flows back into the main memory module inside the PS, i.e. what we called the Memory Loop-Back?

It follows from the background section that each `HPMx` port, say `HPM1`, is reachable under a well defined memory range, say $[S_{HPM1}, E_{HPM1}]$. Whenever the PS produces a read (resp., write) transaction with a physical address A in this range, the `HPM1` port initiates an AXI transaction towards the PL. This activates the AR (resp., AW) channel where the *address* field is populated with A. Similarly to an `HPMx` port, the main memory controller in the PS is associated with its own range of memory addresses $[S_{MM}, E_{MM}]$. But of course the ranges $[S_{HPM1}, E_{HPM1}]$ and $[S_{MM}, E_{MM}]$ cannot be overlapping. Similarly, the PL can initiate a transaction towards main memory via any of the `HPSx` ports.

A first naïve design for a Memory Loop-Back consists in instantiating a direct route from any of the `HPMx` ports, to any of the `HPSx` ports. This is depicted as a dashed line in Figure 1, where `HPM2` and `HPS2` are used. This approach does not work, however, because transactions that leave the PS into the PL will have an address $A \in [S_{HPM2}, E_{HPM2}]$.

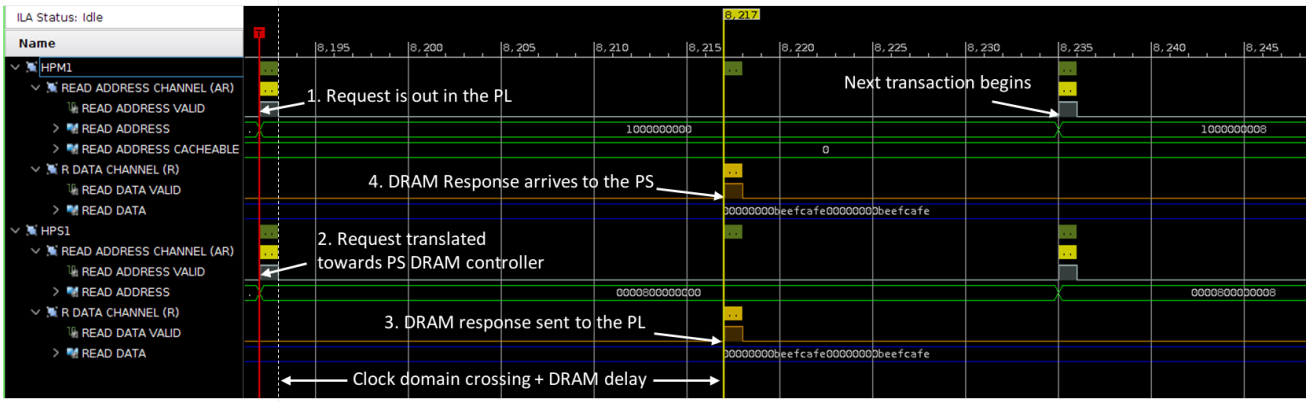

Fig. 2. Cycle-Accurate DRAM Response-Time monitoring using ILA. Translator adds Zero delay.

TABLE II
MEMORY MAP OR RELEVANT SOC RESOURCES

Resource	Description	Start Address	End Address
High DRAM	PS main memory, lower 2 GB	0x00_0000_0000	0x00_7FFF_FFFF
Low DRAM	PS main memory, higher 2 GB	0x08_0000_0000	0x08_7FFF_FFFF
HPM1	HPI port 1 PS-mastered	0x10_0000_0000	0x17_FFFF_FFFF
HPM2	HPI port 2 PS-mastered	0x48_0000_0000	0x4F_FFFF_FFFF
HPM3	HPI port 3 PS-mastered	0x00_8000_0000	0x00_9FFF_FFFF

As they re-enter the PS via the slave port, the address will still be A, which not only is not a valid address in the range $[S_{MM}, E_{MM}]$ for main memory to respond, but it is still in the range $[S_{HPM2}, E_{HPM2}]$. As such, a new transaction is initiated on the HPM2, which restarts the cycle. The result is an infinite loop of incomplete transactions that never produce a response.

A functioning Memory Loop-Back implementation requires minimal logic to manipulate the address of traversing transactions at the PL. For this purpose, we introduce a *Translator* block in the PL, as shown in Figure 1. The Translator defines a slave port connected to HPM1 and a master port connected to HPS1. Before relying requests/responses from the master to the slave port and vice-versa, the Translator performs a simple manipulation on the address bits on both the AR and AW channels. First, the module buffers incoming transactions on its slave port; then it sets the most significant 5 bits to 0 and forwards the transaction on its master port. Responses from main memory are left untouched.

Example: Let us consider the actual address ranges in the SoC of reference reported in Table II. The platform uses 40-bits addresses. The range $[S_{HPM2}, E_{HPM2}]$ = [0x48_0000_0000, 0x4F_FFFF_FFFF] (32 GB), while the range $[S_{MM}, E_{MM}]$ = [0x08_0000_0000, 0x08_7FFF_FFFF] (2 GB)[2]. Before forwarding the transaction on its master port, the Translator modifies bits 36-40 by setting them to 0. As long as the incoming address is in the range [0x48_0000_0000, 0x48_7FFF_FFFF], the outgoing address will fall inside $[S_{MM}, E_{MM}]$.

We implemented the Translator module as a custom IP, because, albeit simple, the described functionality cannot be achieved with the publicly available IP components. We provide a study of the overhead incurred due to transactions being routed via the PL in Section VI-B.

[2]A second block of 2 GB is mapped at [0x00_0000_0000, 0x00_7FFF_FFFF].

C. Transaction-level Inspection

Albeit simple, the Memory Loop-Back provides us with a unique opportunity to inspect the memory traffic generated by an application running on the PS, and the response produced by main memory. To understand the depth of the insights that can be gathered in this way, we instantiated an Integrated Logic Analyzer (ILA). The ILA is essentially a PL block that can be attached to AXI segments to gather a trace of transactions on the SoC that can be later visualized on a host machine. Figure 2 shows the result of an ILA analysis session for a read transaction leaving the PS with address 0x10_0000_0000, and being forwarded to main memory with address 0x08_0000_0000. Not only it can be observed that the transaction is the result of a cache miss, but it is possible to compute the exact round-trip time from the moment it leaves the PS until a response is returned from main memory — highlighted at the bottom of the figure. At this level of analysis, many additional attributes can be observed, such as QoS bits set by the cache controller, memory cacheability, originating processor (or other master), bus saturation conditions, and so on. For clarity, additional signals are not shown in Figure 2.

Being able to achieve such a high degree of inspection in a COTS platform opens up important opportunities. In fact, the important drawback of consolidating hard real-time systems onto COTS multi-core platforms is a lack of knowledge on the exact behavior of the platform, especially when it comes to shared memory hierarchy components. Remarkably, with the PLIM approach, one can perform direct observations on components that had to be previously considered as black-boxes. Moreover, in the next sections of this paper, we demonstrate that PLIM modules can also enact interesting management policies.

IV. CACHE BLEACHING

Having assessed the practicality of the PLIM approach, we present a technique that leverages the PLIM approach and solves long-standing shortcomings of cache partitioning via page coloring. We first provide some background on caches and coloring, then discuss its important shortcomings.

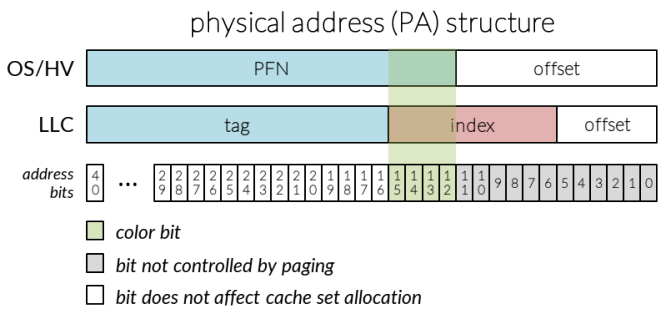

Fig. 3. Bits 12-15 are color bits in considered SoC. There are 16 colors.

Lastly, we present our Cache Bleaching approach and how it addresses the traditional issues with coloring.

A. Background on Caches and Coloring

Consider a multi-core embedded SoC that provides hardware support for virtualization. A single physical addressing space exists where main memory, HPI ports, and I/O devices are mapped and reachable under a range of physical addresses (PA). The memory map for components that are relevant to our discussion is reported in Table II. Second, two stages of memory virtualization are supported by the hardware. The first stage corresponds to virtual addresses (VA), as seen by user-space processes. These 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 is managed by a hypervisor, which operates at a higher privilege level compared to the OS. If no hypervisor is active in the system, then an IPA is translated directly and 1-to-1 to a PA. If enabled, it is the responsibility of the hypervisor to maintain second-stage page tables to allow IPA-PA translations. The MMU handles first-stage and second-stage translations for memory accesses originated by the processors. It follows that a hypervisor can ultimately determine which physical addresses are 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 KB.

Modern embedded SoCs feature multi-level caches. A common cache hierarchy is the following: each processor has a private coherent cache (L1); and all the processors share a second level of cache (L2). The L2 cache is much larger in size than the private L1. The L2 is often also the last-level cache (LLC). Although this is the case for the SoC used in our evaluation, what presented in this work directly applies to SoCs with additional cache levels. A miss in LLC causes an access to main memory. Generally, each level operates as a *set-associative* cache with associativity W. A cache with total size C_S and associativity W is structured in W ways of size $W_S = C_S/W$ each. In the considered SoC, $C_S = 1$ MB, W = 16, and $W_S = 64$ KB.

Caches do not store individual bytes. Instead, they store multiple consecutive bytes at a time, forming a *cache line* of L_S bytes each. In our SoC, $L_S = 64$ 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. When a *cacheable* memory location is accessed by a processor, the value of its address determines which cache location to look-up or allocate in case of a cache miss. Addresses used for cache look-ups can be PAs or VAs. In shared caches (e.g. L2/LLC), however, both index and tag bits are from PAs. For this reason, they are said to be *physically-indexed*, *physically-tagged* (PIPT) caches. The least-significant bits of the PA encode the specific byte inside the cache line. 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*. The 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, which in our SoC correspond to PA bits 6-15. A PA 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. A breakdown of PA bits for the SoC used in this work is provided in Figure 3.

Recall that a 4 KB page is the finest granularity at which an OS/hypervisor can manage VA/IPA/PA addresses. For instance, each page spans through 64 cache lines when $L_S = 64$ bytes. If W_S is larger than 4 KB, then multiple pages with consecutive PAs can be simultaneously stored in the same cache way. Take C as 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 C_i is cached, its lines can never evict cache lines that belong to a page with color C_j, as long as $C_i \neq C_j$. This principle is at the base of color-based cache partitioning [1], [7]–[9]. In the SoC under analysis, $C = 16$ and the PA bits that affect the color of a page are bits 12-15 — see Figure 3.

B. Shortcomings of Cache Coloring

Albeit powerful, extensively studied and adopted, page coloring comes with significant shortcomings.

Main Memory Waste: If page coloring is used to perform cache partitioning, only a subset of C consecutive physical pages can be assigned to an application/VM. For instance, with $C = 16$, consider the creation of two differently sized partitions for two VMs. VM-1 is assigned 2 colors, while VM-2 is assigned 8 colors. With this scheme, out of every 16 pages with consecutive PAs, the first 2 are mapped to VM-1 and the last 8 to VM-2. The fact that only a subset of PAs can be allocated to each VM implicitly limits the amount of main memory that each VM can address. If we had 2 GB of main memory, in our example VM-1 could only use 256 MB, while VM-2 1 GB. This is far from ideal because larger LLC partitions should be allocated to VMs with LLC sensitive applications; while smaller LLC partitions to applications having a memory footprint too large to fit in LLC. But large-footprint applications might not run with only 1/4

of main memory. At the same time, 1/2 of main memory is wasted on applications that require only a fraction of it.

Recoloring Overhead: Cache partitioning via coloring is a lightweight operation if performed at boot or task creation time. But once colored pages are in use, they contain instructions/data used by an application/OS to function. As such, performing a runtime change in the partitioning scheme is a costly operation because the content of a massive number of physical pages needs to be relocated. And yet, dynamically changing the size of cache partitioning is a useful operation as a system undergoes mode changes — e.g. in case of a criticality-level switch in a mixed-criticality system [10], or in reaction to new mission-level objectives where the application workload changes.

C. Address Bleaching

Performing address bleaching means to re-compact an addressing space to lose coloring information. In our design, this operation is done in a PLIM module after cache look-up, but before transactions reach main memory. The result is that a shared cache can be partitioned using page coloring, albeit the corresponding data in main memory are stored strictly sequentially, i.e. without holes.

Consider a system with C total colors (and hence $\log_2 C$ color bits) and a colors-to-VM assignment of the form $\mathbf{C} = [C_l, C_h]$. For every C consecutive pages aligned at the C-th page boundary, the VM is mapped pages with colors C_l through C_h, with $C_l, C_h \in \{0, \ldots, C-1\}$ and $C_l \leq C_h$.

Example: Take $C = 16$ and the coloring specification $\mathbf{C} = [2, 3]$. A VM mapped starting from base $B =$ `0x00` will be given page frame numbers (PFN)[3] `0x02`, `0x03`, `0x12`, `0x13`, `0x22`, and so on.

If the coloring specification $\mathbf{C}$ as well as the base address B of the mapping is known and aligned to the C-th page boundary[4], it is possible to efficiently construct a contiguous non-colored mapping starting by only reasoning on (1) the base address B of the mapping; (1) the address $A \geq B$ to be bleached; and (3) the coloring specification $\mathbf{C}$. The resulting bleached address is $\bar{A}$ and can be computed as follows, considering B, A, and $\bar{A}$ expressed as PFNs.

$$\bar{A} = A - ((A - B)/C) \times (C - C_h - C_l + 1) - C_l. \quad (1)$$

In Equation 1, "/" and "×" indicate integer division and multiplication, respectively. From the example presented above, if $B =$`0x00` and $A =$`0x22`, then $\bar{A}$=`0x04`, and indeed A was the fifth page of the colored mapping.

Equation 1 is already efficient to compute in the PL because it involves only integer operations. But the bleaching operation can be optimized even further. In particular, when the coloring specification is *well-aligned*, bleaching can be performed by simply "dropping" a portion of the colored address bits. To be well-aligned, the following must hold: (1) the number of assigned colors $(C_h - C_l + 1)$ is a power of 2; and (2) the value of C_l is multiple of $(C_h - C_l + 1)$. When a specification is well-aligned, then dropping the most significant $\log_2(C) - \log_2(C_h - C_l + 1)$ color bits will return the bleached address.

[3]The difference between a PA and a PFN is simply that the least significant 12 bits encoding the offset within a 4 KB page are omitted.

[4]The rules for bleaching can be trivially extended to consider base addresses that are not C-th page-aligned, but for simplicity we keep this assumption.

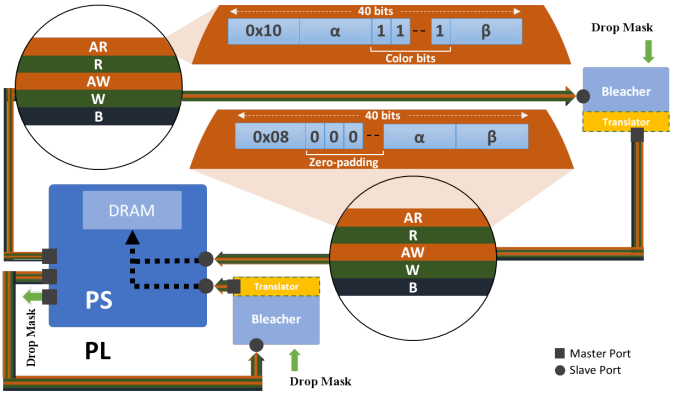

Fig. 4. Bleacher performs de-coloring on AR and AW channels. Only AR is magnified; AW is similar.

Example: The following are some cases of well-aligned color specifications: [0, 3], [0, 15], [4, 11], [8, 11]. Among these, consider specification [8, 11]. And once again, take $B =$ `0x00`. Hence, the VM will be mapped at PFNs `0x08`—`0x0B`, `0x18`—`0x1B`, `0x28`—`0x2B` and so on. To recover the bleached PFN $\bar{A}$ for A= `0x29`, we can drop the most significant 2 bits. The result is $\bar{A}$= `0x09`, and indeed A was the 10-th page of the colored mapping.

D. Bleacher Design

We designed a PLIM module, namely the *Bleacher* to perform the optimized bleaching transformation discussed above. Alike the Translator discussed in Section III, the Bleacher module is placed between a `HPMx` and `HPSx` port. Moreover, it exposes a memory-mapped configuration space through which a hypervisor can specify the type of bleaching to perform. The current implementation only supports well-aligned color specifications, and hence the only piece of configuration required is the number of color bits to drop. In our prototype, the Bleacher accepts a bitmask with as many bits as the number of colors bits in the system — four, in the considered SoC. If any of the bits of the mask is set to 0, the corresponding bits in the address of a read/write AXI transaction arriving from the PS is dropped. The exact operation performed by the Bleacher is depicted in Figure 4. From a 40-bit address, an upper portion α and a lower portion β are identified. Any bit between α and β is dropped, and padding is inserted as needed. Finally, the address is re-based to target high DRAM. The logic of the Bleacher was implemented with highly parallel logic. The resulting PLIM module introduces only two clock cycles of delay in forwarding a transaction with the bleached address.

To exploit additional available parallelism, in our final design, depicted in Figure 4 we instantiated two Bleacher modules, attached to the pairs of ports `HPM1` and `HPS1` (Bleacher 1), and `HPM2` and `HPS2` (Bleacher 2). Having two Bleachers also allow defining two independent coloring/bleaching schemes on each `HPM` port.

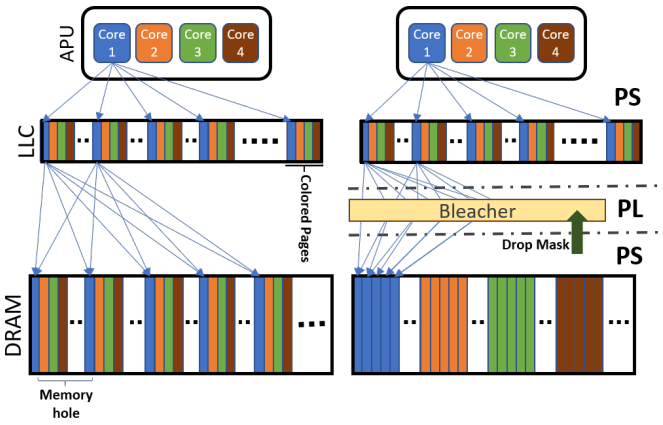

Fig. 5. With the Bleacher (right), colored addresses are de-colored (bleached) hence, previously scattered pages (left) become contiguous in DRAM

With the Bleacher in place, it is now possible to define colored mappings that span through an addressing range well beyond the available physical memory, but that always results in contiguous accesses in main memory. This is exemplified by Figure 5, where in terms of address transformation, it can be noted that the Bleacher acts as a *magnifying lens* on the physical addresses seen by the LLC. As such, the location of pages in main memory is **entirely decoupled** from the way they map in LLC, which is the ultimate goal of bleaching. We evaluate a full system setup with colored/bleached VMs in Section VI.

V. Zero-Copy Recoloring

The unique capability provided by the Bleacher (see Section IV-D) is that the rule according to which bleaching is performed can be changed instantaneously at run-time. This is the premise for zero-copy re-coloring. We hereby detail a prototype implementation of a partitioning hypervisor (Jailhouse) capable of performing zero-copy re-coloring of VMs.

A. Background on the Jailhouse Partitioning Hypervisor

In this work, we consider cache partitioning between concurrent VMs in a way that each VM is unaware of the partitioning. As such, we consider the case where page coloring is performed at the second stage of memory translation, i.e., between IPAs and PAs (see Section IV-A). To enforce coloring in this way, one can leverage a lightweight partitioning hypervisor. A similar approach has been followed in [7], [9], [11], [12], that have considered Xvisor [13], KVM [14] and Jailhouse [15], respectively.

For this work, we have used the open-source partitioning hypervisor Jailhouse that already supports the definition of colored mappings for VMs[5]. One of the main advantages of Jailhouse is that it only partitions SoC resources — i.e., CPUs, memory regions, I/O devices — but it does not perform any VCPU scheduling, and it allows VMs to interact with (allocated) I/O devices directly. The Jailhouse hypervisor can

[5]The source code is available at https://github.com/siemens/jailhouse.git under the branch `wip/cache-coloring`.

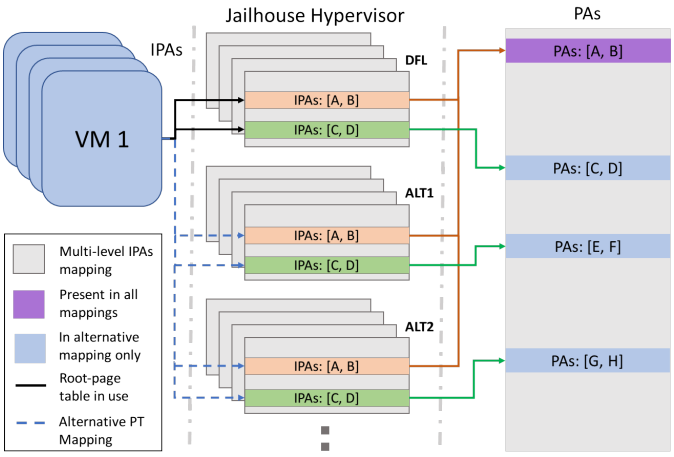

Fig. 6. The hypervisor maintains a default (DFL) IPA→PA mapping, as well as multiple alternative ones (ALT1, ALT2, ...).

activate multiple VMs on non-overlapping sets of processors — e.g., a Linux VM, a bare-metal VM. Each VM is unmodified and sees a contiguous space of IPAs. At the second stage, Jailhouse maps IPAs of different VMs to PAs with a configurable color specification.

B. Alternative Mappings

In this work, we have extended the Jailhouse hypervisor to support zero-copy re-coloring when operating in conjunction with Bleacher modules in the PL.

The first extension required to support zero-copy re-coloring is to support the definition of multiple sets of IPA→PA mappings for the same range of IPAs. A traditional partitioning hypervisor maintains a single set of multi-level paging structures to map IPAs to PAs for a VM. In these paging structures, the top-level page is called the *root-page*. In order to install a mapping, the Hypervisor installs a pointer to the root-page in the MMU(s) of the processor(s) assigned to the VM.

We have extended Jailhouse to maintain multiple sets of paging structures, and hence multiple root-pages. Each mapping different from the one used to bootstrap the VM is called an *alternative mapping*, while the mapping used at bootstrap is called *default mapping*. Note that in each alternative mapping, some portions of the mapped IPA ranges do not change in terms of corresponding PAs. In our setup, only one single IPA range is mapped to alternative PAs, albeit there is no limit on how many alternative mappings can be defined for the same IPA range. For instance, Figure 6 depicts the case where a VM is allocated two ranges of IPAs, of which one has three alternative mappings. Our modified Jailhouse bootstraps a VM using its default mapping. At the same time, it constructs and maintains multiple paging trees, one for each configured alternative mapping. The first level of each alternative mapping is a page called the *root-table*.

C. Recoloring via Alternative Mappings and Bleaching

Being able to define alternative mappings and to perform configurable address bleaching enables zero-copy re-coloring. In a nutshell, consider a VM that is allocated 1 GB of main

memory. This will correspond to a 1 GB range of IPAs. The default mapping for these IPAs can be taken as a contiguous range of PAs mapping directly to the main memory. At the same time, one or more alternative mappings for the same 1 GB IPA range is maintained, such that (1) alternative mappings have different coloring configurations; and (2) the base physical address of the mappings are adjusted to be in the range of the Bleacher module.

With the aforementioned setup in place, all it takes to recolor a VM is (1) a switch of the root-table pointer in the MMU(s); and (2) reconfiguration of the Bleacher module with the new bit drop mask, as discussed in Section IV-D. We implemented this capability in the modified Jailhouse that can be triggered at run-time on a VM. More in detail the steps to perform a runtime mapping switch are the following: (1) suspend the target VM; (2) find root-page pointer of target mapping; (3) Perform a clean and invalidate operation on private and shared caches; (4) Invalidate the relevant entries in the TLBs of the target processors; (5) Install new root-page table pointer in the MMUs of the target processors; (6) Fetch base and compute drop mask for target mapping; (7) Send the drop mask for the new mapping to the Bleacher module; and finally (8) Resume the target VM.

Let us take a concrete example, which is in line with the setup used in our experiments. We consider a Linux VM that is allocated 1.5 GB of main memory. This corresponds to an addressing range of size `0x6000_0000`. In the default mapping, non-colored PS main memory is assigned starting from address `0x08_1000_0000` (see Table II), ending at page `0x08_6FFF_F000`. This also corresponds to the range of IPAs under which the VM sees the 1.5 GB aperture.

An alternative mapping is defined, referred to in Section VI as CASE1, where a single cache partition (color) is assigned to the VM. We use color specification $\mathbf{C_1} = [15, 15]$ and use the Bleacher on `HPM1`. As such, the base of the mapping is `0x10_1000_0000`. But because only one page every 16 is mapped to a PA, the last mapped page starts at PA `0x16_0FFF_F000`. Note that without the Bleacher and with the same base address and coloring specification, the VM would be limited to 1/16 of 1.5 GB, i.e., to 96 MB of memory.

With $\mathbf{C_1}$, the drop mask can be computed as `0x0` i.e., all the color bits (12-15) will be dropped at the Bleacher. When the alternative mapping and drop mask are installed by Jailhouse, the result of bleaching on the last mapped page can be computed as follows. Consider the distance from the base address `0x16_0FFF_F000 - 0x10_1000_0000 = 0x05_FFFF_F000`. Next, bits 12-15 are dropped, so the non-colored distance from the base is `0x00_5FFF_F000`. Finally, the address can be added to the base, and the top bits replaced to target the higher 2 GB of PS DRAM (MM_HI in Table II), as done in the Translator (see Section III-B). The resulting address sent to the main memory is `0x08_6FFF_F000`, i.e., the same as in the default mapping.

It follows that the same exact content in the main memory will be accessed, with the important difference that when going through the alternative mapping, only one cache color will be occupied by the VM. This demonstrates that **zero-copy recoloring** can be performed. Moreover, it makes a second important point. That is, by keeping an alternative mapping that bypasses the PL, one does not have to commit to using any of the PLIM modules. For instance, if the overhead introduced by PLIM modules is too deemed too high, or no contention on cache is expected at a certain point in time, it is always possible to dynamically reroute a VM's traffic directly to main memory. We provide an evaluation of the overhead for alternative mapping switching in Section VI.

VI. EVALUATION

We now conduct a set of experiments to understand different aspects of the implemented design. We hereby try to answer the following questions.

1) What is the overhead introduced by PLIM modules on individual transactions and on the runtime of applications? This is investigated in Section VI-B.
2) Is performing coloring using our PLIM approach beneficial for performance isolation? This question is approached in Section VI-C;
3) What is the overhead to perform zero-copy dynamic recoloring? Section VI-D explores this aspect; and
4) Is it possible to dynamically manage the trade-off between performance and cache occupation via zero-copy recoloring? This is discussed in Section VI-E.
5) What is the overhead in terms of FPGA resources and additional power consumption of the presented PLIM modules? This dimension is explored in Section VI-F.

A. Experimental Setup

We have performed a full system implementation on a Xilinx ZCU102 development system, featuring a Xilinx Zynq UltraScale+ XCZU9EG SoC. We implemented our system with two Bleacher PLIM modules responding under the PA address ranges of `HPM1` and `HPM2`, respectively — see Table II. Their drop mask configuration interfaces are mapped under `HPM3` at `0x8000_0000` and `0x9000_0000`, respectively. We deploy four VMs, each statically allocated to one of the four ARM Cortex-A53 CPUs. VM1 runs an instance of Linux 4.9 and it is used to run our benchmarks. We study the behavior of all the San Diego Vision Benchmarks (SD-VBS) [16]. VM2-4 run synthetic memory-intensive applications, a.k.a. *mem-bombs* used to produce cache interference. Each mem-bomb performs an infinite loop of write operations sequentially over a buffer of a size larger than the LLC (1.5 MB).

We consider 6 different configurations for the VMs under analysis, named CASE0 through CASE5. A summary of the colored alternative mapping (and corresponding cache partitioning) used for each VM in each case is provided in Table III. For each case and VM i, the table reports the coloring specification $\mathbf{C_i}$, and the resulting number of cache partitions #P. As can be seen from the table, CASE0 is the case where there is no isolation; in cases 1 through 4, the VM under analysis has 1, 2, 4, or 8 dedicated partitions. Finally,

TABLE III
EVALUATION CONFIGURATIONS

CASE	VM1		VM2	
	C_1	#P	C_2	#P
0	0-15	**16**	0-15	16
1	15-15	**1**	14-14	1
2	14-15	**2**	12-13	2
3	12-15	**4**	8-11	4
4	8-15	**8**	0-7	8
5	0-15	**16**	15-15	1

CASE	VM3		VM4	
	C_3	#P	C_4	#P
0	0-15	16	0-15	16
1	0-7	8	0-7	8
2	0-7	8	0-7	8
3	0-7	8	0-7	8
4	0-7	8	0-7	8
5	15-15	1	15-15	1

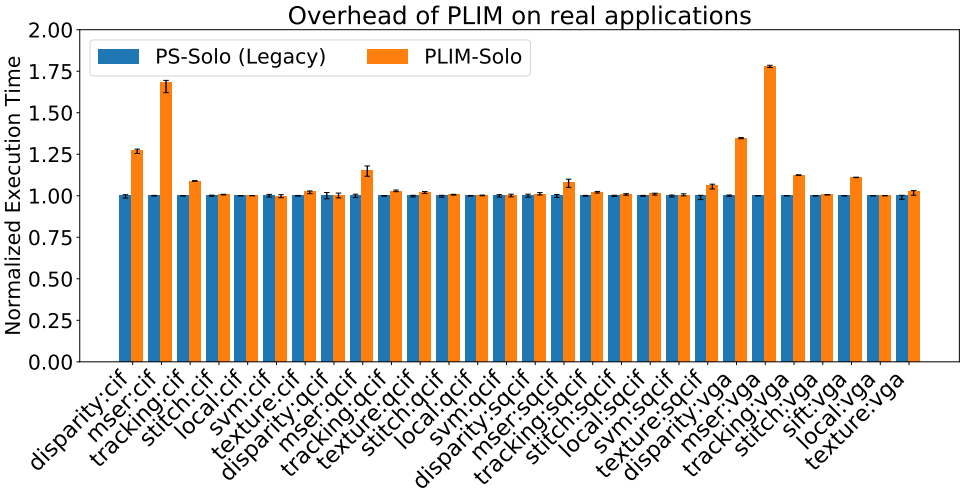

Fig. 7. Overhead of PLIM on San-Diego Vision Benchmarks with different input sizes.

in CASE5 the VM under analysis has 16 partitions, of which 15 private and 1 shared with the other VMs.

B. Memory Loop-Back and Bleacher Overhead

Exploiting comprehensive information provided by the ILA, we can observe the temporal response of each transaction at the granularity of PL cycles. Figure 2 shows an ILA export, triggered on a single read transaction. Note that this ILA has two probes; one right after entering the PL and another before going back into the PS (as shown in Figure 1). This way, an AXI read transaction is captured twice, once when sending the desired address to the DRAM via the AR channel and a second time when data are delivered by the DRAM back to the requesting core via the R channel.

The ILA's second probe is instantiated on the HPS1 port and receives the translated request immediately on the very same clock. The DRAM controller then takes 25 cycles (about 83 ns) to produce a response. As confirmed by the behavior of the signals on HPM1 and HPS1, the introduced Translator adds no delay. For address bleaching, additional combinatorial logic is required. In our current design, the Bleacher module introduces 2 PL clock cycles of delay on each transaction. We also observed an additional memory latency compared to the case when transactions reach the DRAM without going through the PL. A technical consultation with specialists at Xilinx revealed that the extra delay is due to a transaction performing multiple clock domain crossings (CDC) when going through the PL. The CDC impacts each transaction multiple times. Consider for instance a read transaction. The CDC is paid four times: (1) when the AR command leaves the PS to enter the PL; (2) when the AR command is relied to the DRAM from the Translator block in the PL; (3) when the response on the R channel leaves the DRAM to enter the PL; and (4) when the PL forwards the response on the R channel back to the PS. While CDC introduces a visible performance hit on some applications, there are two main reasons leading us to believe that CDC delays will decrease in future-generation PS-PL SoCs. First, FPGA frequency increases naturally with newer generations. As the frequency gap between CPUs and FPGA fabric shrinks, the CDC delay also decreases. Second, next-generation PS-PL SoCs are designed around the idea of a tighter collaboration between non-configurable components and PL. For instance, the announced Xilinx Versal SoCs [17] include heterogeneous processing cores and high-speed DRAM memory controllers surrounded by FPGA fabric to define scratchpads and data engines. A low CDC delay is key for platforms like the Versal and for future platforms that will follow the same paradigm.

Next, we study the actual impact of the added overhead on real applications in Figure 7. We use the same SD-VBS applications used in the rest of our evaluation. In the figure we consider two cases. In both cases, the application under analysis executes alone in the system. In the first one (PS-SOLO), however, no memory traffic re-routing is enacted. Conversely in the second case (PLIM-SOLO), the application's memory traffic is routed through the PL and via a Translator block (see Section III-B). Average, maximum and minimum observed runtimes are reported in each bar. From the figure, it emerges that apart from a small number of cases, real applications are only slightly affected by the extra PL overhead. The reason why some applications suffer a higher slowdown when their memory traffic is routed through the PL is twofold. On the one hand, some applications generate less DRAM traffic. On the other hand, when no data dependencies exist between a memory transaction and the subsequent instructions, the micro-architecture is able to (partially) hide the extra delay introduced by CDCs. This is done by reordering instructions and data fetches. Therefore, applications that exhibit a memory access pattern with a higher degree of data dependencies are more sensitive to an increased per-transaction latency.

C. Performance Isolation

In order to study the effect of cache partitioning on the SD-VBS applications, we compare a number of configurations. As our performance baseline, we consider the PS-SOLO case, where each application executes in the most favorable conditions, i.e. alone in the system with no cache partitioning and

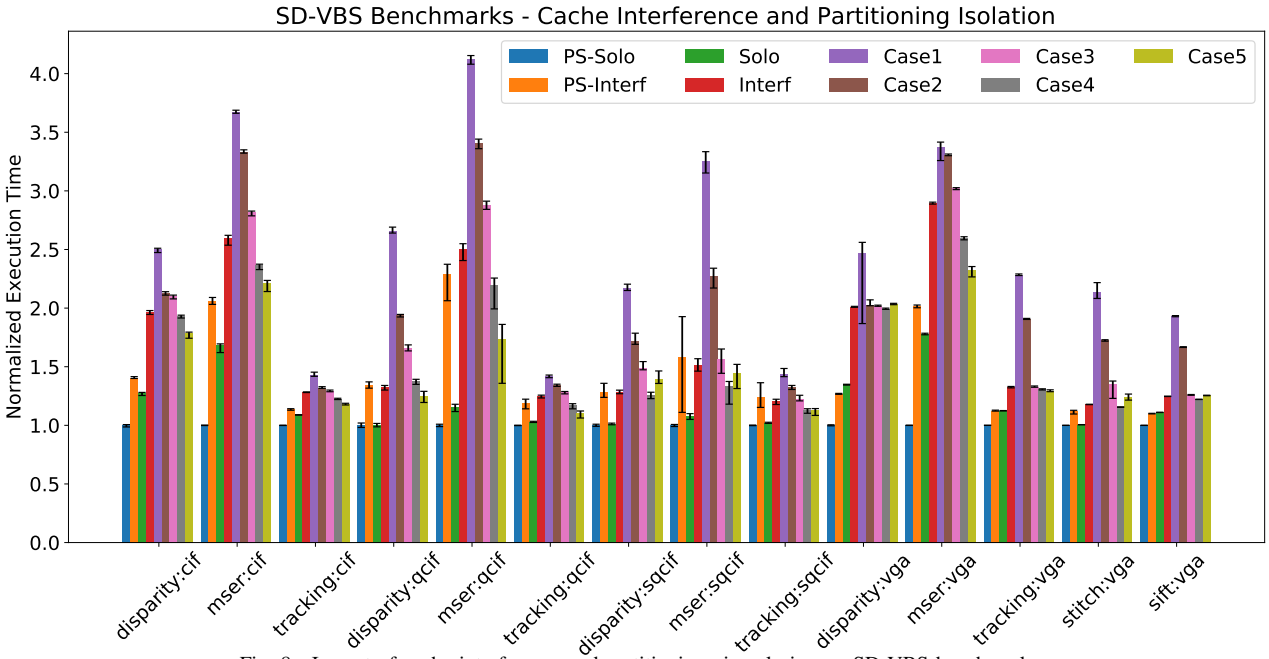

Fig. 8. Impact of cache interference and partitioning via coloring on SD-VBS benchmarks.

without going through the PL. Next, in the PS-INTERF, each benchmark contends for cache space with 3 mem-bombs active on the other cores but with no partitioning and without going through the PL. In all the remaining cases, memory traffic is always routed through the PL. In the SOLO experiment each benchmark is executed alone on the platform with full cache assignment, i.e. we use configuration CASE0 where VM2-4 are kept idle. Next, we perform a run with full interference, namely INTERF using configuration CASE0 with mem-bombs active on VM2-VM4. While keeping the interfering VMs active, we then go through the remaining cases 1 to 5 detailed in Table III. The SD-VBS benchmarks come with multiple input sizes. We performed the experiments on all the input sizes, but report only the results for the intermediate sizes where the benchmarks are LLC sensitive. These are `cif`, `qcif`, `sqcif`, and `vga`. A more detailed description of the available input sets is provided in [16]. The obtained results for a subset of benchmarks are reported in Figure 8. The selection of benchmarks reported in Figure 8 was done merely based on how interesting are the obtained results. For completeness, Figure 9 reports the reminder of the plots. Each cluster of bars reports the averaged result of 30 runs, with one bar per system configuration; the error bars report the maximum and minimum execution times observed across all the runs. The label at the bottom of each cluster reports which benchmark the cluster refers to and on which input set the runs were performed, in the format `benchmark:input`.

A few important remarks can be made about Figure 8. First and non-surprisingly, different benchmarks are impacted differently from partitioning. For instance, in the cases `disparity:cif`, `mser:cif`, `disparity:qcif`, and `mser:qcif`, assigning 1 private cache partition (CASE1) as opposed to 4 (CASE4) drastically reduces the slowdown suffered due to interfering workload from $2\times$ to $1.5\times$, from $2.2\times$ to $1.4\times$, from $2.6\times$ to $1.3\times$, and from $3.7\times$ to $1.9\times$, respectively. Therefore, zero-copy recoloring represents a powerful capability that enables adjusting cache allocation in response to changes in the application workload. Second, by comparing the second bar of each cluster (FULLINTERF) with the sixth (CASE4) and seventh bar (CASE5), it can be noted that LLC interference can be mitigated by enacting cache partitioning via the proposed Bleacher module, the most dramatic case being `mser:qcif`. In this case, the benchmark suffers a slowdown of $2.2\times$ under contention, which, by restricting the cache partition for the interfering workload (CASE5), is reduced to $1.3\times$. At the same time, if a benchmark is mostly bottle-necked by main memory bandwidth, then restricting the number of partitions available to interfering workload does not help and can have the opposite effect. The latter is the case for `disparity:sqcif` and `disparity:vga`. Third, by comparing the cases reported in Figure 9, we observe that when an application is not LLC sensitive, assigning a not-too-small cache partition — larger than one color — is enough to isolate the benchmark. In fact, the slowdown observed under CASE1 is due to this SoC featuring physically-indexed L1 caches [5] with a way size of 2 pages. Thus, assigning only one partition leads to implicit partitioning of the private L1.

D. Recoloring Overhead

Here, we investigate the time overhead required to perform an alternative mapping switch and necessary cache invalidation to perform zero-copy recoloring. We measure the overhead of the overall root-page table switching procedure with increasing sizes of main memory allocated to a VM under analysis. The result is reported in Figure 10. Each data sample is the aggregation of 200 samples, with error bars capturing the standard deviation of the measurements. As can be seen

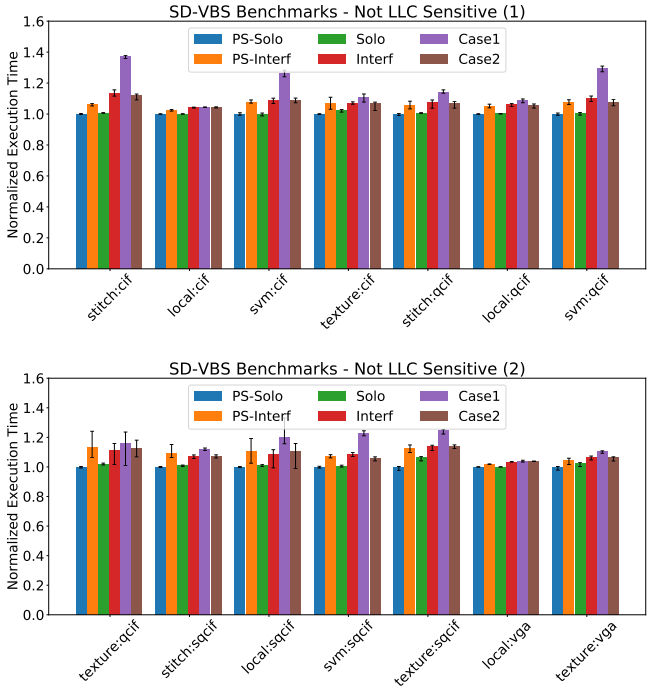

Fig. 9. Impact of cache interference and partitioning via coloring on SD-VBS

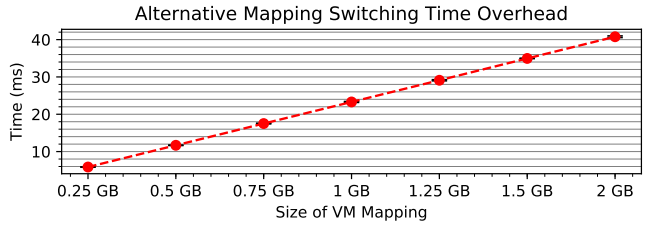

Fig. 10. Alternate mapping time switching overhead for a target VM with increasing amount of mapped main memory.

from Figure 10, the time required to perform the switch is linear with the size of the mapping. This is because, in ARMv8 family processors like the Cortex-A53, cache cleanup&invalidation (C+I) can be carried out by providing a cache way/set coordinate for the line to be invalidated, or by providing a VA for which the corresponding PA needs to be invalidated. In our implementation, we perform C+I via VAs. This has the advantage of not invalidating cache lines belonging to other VMs, and hence only impacting the VM being recolored. Moreover, C+I through VAs is ensured to be broadcasted to all the processors that may have dirty local copies. In this sense, it is safer to perform compared to way/set-based invalidation. The drawback of VA-based C+I is that the hypervisor needs to loop through all the VAs assigned to the target VM to complete the operation. It follows that what depicted in Figure 10 is an upper-envelope of the switching overhead without any optimization on the amount of memory being invalidated. It can be drastically reduced by limiting the invalidation to the portion of memory actually mapped to applications, as opposed to the entire VM's main memory. Moreover, way/set invalidation can be used to significantly speed-up the switch when it is known that no data is shared between the target VM and the other VMs.

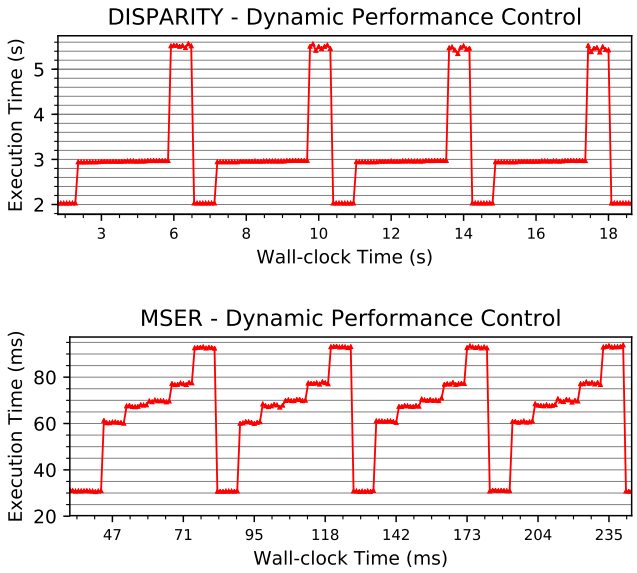

Fig. 11. Dynamic performance control of the benchmark `disparity` on input `vga` (top) and `mser` on input `qcif` (bottom) via runtime re-coloring.

E. Dynamic Performance Management

Support for alternative mapping switching provides the ability to dynamically adjusting the trade-off between performance and LLC utilization of applications. Additionally, if traffic management via PLIM is deemed unnecessary at runtime, it is possible to exclude the PL route and direct memory traffic of VMs directly to main memory. The same goes for cases in which the overhead of going through the PL is unacceptable. To evaluate this aspect, we perform the setup of a Linux 4.9 VM mapped to 1.5 GB of main memory and 6 total mappings (1 default + 5 alternative mappings). The first mapping defines a direct route to PS main memory; the second through the fifth are colored mappings via the Bleacher module with 16, 8, 4, 2 and 1 cache partitions, respectively. We execute the same benchmark continuously on the VM and rotate through the mapping assignments at runtime, roughly every 8 runs of the benchmark. The results for benchmarks `disparity` (on input `vga`) and `mser` (on input `qcif`) are reported in Figure 11 (top) and Figure 11 (bottom). From these results, we conclude that zero-copy recoloring can, in fact, be used to (1) study the sensitivity of applications to the quota of LLC assigned; and (2) to manage applications' performance online.

F. Area and Power Analysis

Table IV provides a breakdown of power consumption and FPGA resources used to implement the designs discussed in this paper. The first design considered in the table has a single Bleacher module that includes a Translator block. The second design only includes a Translator block. This block performs just re-wiring of AXI channel signals. As such, it does not consume any FPGA resources nor additional power. The last design reported in the table is a full system with two Bleacher modules. The design also includes a Xilinx SmartConnect [18] interconnect to interface multiple PS master ports to the same

TABLE IV
FPGA AREA AND POWER CONSUMPTION

Design	Block Name	Power (W)	FFs	LUTs
Bleacher+ Translator (single)	PS	3.195 (81.32%)	-	-
	Bleacher+Translator	0.009 (0.23%)	678 (0.12%)	519 (0.19%)
	Total On-Chip	3.929		
Translator Only	PS	3.198 (81.52%)	-	-
	Translator	0.000 (0.00%)	0	0
	Total On-Chip	3.923		
Full System: 2x Bleacher+ Translator 1x SmartConnect	PS	3.199 (79.28%)	-	-
	Bleacher+Translator 1	0.004 (0.10%)	652 (0.11%)	311 (0.11%)
	Bleacher+Translator 2	0.004 (0.10%)	652 (0.11%)	311 (0.11%)
	SmartConnect	0.101 (2.50%)	7326 (1.33%)	7043 (2.5%)
	Total On-Chip	4.035		

Bleacher. Overall, it can be noted that the power draw of the Bleacher and Translator modules is negligible compared to the power consumed by the PS subsystem. The quota of FPGA resources consumed by PLIM modules is also a small fraction of the available total.

VII. RELATED WORK

This work tackles the problem of achieving predictability in multi-core SoCs via a co-design of software and hardware techniques to enforce fine-grained control over shared hardware resources. We hereby provide a brief survey of the most closely related literature.

Hardware Re-design: a first category of works proposed clean-slate re-designs of multi-core platforms to achieve better and more predictable performance [19]–[21]. Other works have focuses on individual components known to represent main sources of unpredictability. Hardware modifications to implement different schemes cache partitioning schemes were proposed in [22]–[27]. Advanced schemes that use Zcaches [28] to implement more fine-grained partitioning were proposed in [29], while utility-driven schemes were explored in [30]. Cache designs that provide hard real-time guarantees were explored in [22], [26], [31]. Hardware adaptations for cache locking have been studied in [32], [33]. Similarly, hardware-level modifications to DRAM controllers targeting systems with constraints on tail latency have been studied [34]–[40]. The work in [41] combines modifications to caching strategies and DRAM controllers while reusing traditional multi-core CPUs.

The core philosophy of the proposed PLIM approach is that an improved level of inspection and control is possible by directly intercepting data-flows between processors, memory and I/O resources. In this sense, the work on class-of-service based QoS architecture, namely CoQoS [42], comes close to the PLIM approach. It proposes architectural modification to tag data transactions. Tags are then used to perform fine-grained management. QoS enforcement at the SoC level was also proposed in [43]–[47]. A more encompassing work that also considers I/O devices and the capability of defining a control plane in software was proposed in PARD [48]. Unfortunately, hardware-based approaches are not immediately applicable to COTS platforms. Compared to these works, PLIM leverages existing hardware capabilities to define data-flow based manipulation primitives.

Software-based Resource Management: software-based techniques have been explored to reduce the pessimism in the presence of non-deterministic hardware. Cache partitioning and page coloring are often used to manage shared CPU caches [8], [49]–[52]. In [53], different page colors are used for applications' and OS's dynamic data. The work in [54] mentions the possibility of performing cache bleaching by physically re-wiring the CPU-to-DRAM signals so to drop a subset of color bits. The approach was not implemented and has two main limitations. First, it is limited to what we defined as *well-aligned* color specifications; second, no run-time reconfiguration is possible. Our PLIM-based approach demonstrates the practicality of cache bleaching in PS-PL SoCs and overcomes said limitations. Cache locking has been largely investigated using a combination of (existing) hardware support and software management [1], [55]–[62]. A technique to perform deterministic cache content allocation in shared caches with random replacement policy was presented in [9]. Similar mechanisms have targeted and integrated the software-level management of multiple resources [10], [63]–[66]. On platforms that feature scratchpads, a few works have proposed techniques to leverage local memories to relieve contention in main memory [67]–[69]. The work in [12] is the most closely related because it defines a bus translator block to mitigate the unacceptable memory waste introduced due to page coloring on small-sized scratchpad memories.

Our work is also concerned with fine-grained observability of memory traffic between processors and memory hierarchy. In this sense, two closely related works are [70], [71]. In the former, FPGA logic is connected to the trace port of a target SoC to capture traces of instructions; in the latter a simulated study was conducted on the ability of tracing memory operations in partially re-configurable SoCs.

What sets this work apart from the literature surveyed above is that (1) with the PLIM approach we postulate the possibility of using programmable logic for inspection and control of memory traffic between embedded processors and main memory; (2) we demonstrate that PLIM can solve long-standing issues with page coloring; (3) we provide a full system design to enact PLIM on real applications.

VIII. CONCLUSION & FUTURE WORK

In this work we have explored the use of PL to implement management modules that sit in-between processors and main memory. This approach, namely PLIM, enabled an unprecedented level of control and inspection of memory traffic in COTS SoCs. As a case study, we have proposed and implemented Cache Bleaching: a strategy to solve long-standing issues of page coloring based cache partitioning.

As part of our future work, we plan to leverage PLIM to study application-aware memory scheduling primitives; to enact online profiling and prediction of memory access patterns; and to carry out anomaly detection and to enforce complex security policies.

REFERENCES

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

[2] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 32:1–32:36, Nov. 2015. [Online]. Available: http://doi.acm.org/10.1145/2830555

[3] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proc. of the 4th ACM EuroSys*. USA: ACM, 2009, pp. 89–102.

[4] Xilinx, Inc., "Zynq UltraScale+ MPSoC - All Programmable Heterogeneous MPSoC," August 2016. [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[5] ARM, *ARM Cortex-A53 MPCore Processor Technical Reference Manual*, ARM, 2016.

[6] ——, *AMBA AXI and ACE Protocol Specification*, ARM, 2011.

[7] P. Modica, A. Biondi, G. Buttazzo, and A. Patel, "Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms," in *2018 IEEE International Conference on Industrial Technology (ICIT)*, Feb 2018, pp. 1651–1657.

[8] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical os-level cache management in multi-core real-time systems," in *Proc. of the 25th ECRTS*, 2013, pp. 80–89.

[9] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, "Deterministic memory hierarchy and virtualization for modern multi-core embedded systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2019, pp. 1–14.

[10] 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-critcality provisioning," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–12.

[11] H. Kim and R. R. Rajkumar, "Real-time cache management for multi-core virtualization." ACM Press, 2016, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2968478.2968480

[12] G. Gracioli, R. Tabish, R. Mancuso, R. Mirosanlou, R. Pellizzoni, and M. Caccamo, "Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 27:1–27:25. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10764

[13] A. Patel, M. Daftedar, M. Shalan, and M. W. El-Kharashi, "Embedded hypervisor xvisor: A comparative analysis," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 682–691.

[14] C. Dall and J. Nieh, "Kvm/arm: the design and implementation of the linux arm hypervisor," vol. 49, 02 2014, pp. 333–348.

[15] J. Kiszka, V. Sinitsyn, H. Schild, and contributors, "Jailhouse hypervisor," Siemens AG on GitHub, https://github.com/siemens/jailhouse, 2018, accessed: 2018-03-31.

[16] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 55–64.

[17] Xilinx, "Versal: The First Adaptive Compute Acceleration Platform (ACAP)," https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf, 2018, [Online; accessed 16-January-2019].

[18] Xilinx, "Smartconnect v1.0 — logicore ip product guide," https://www.xilinx.com/support/documentation/ip_documentation/smartconnect/v1_0/pg247-smartconnect.pdf, March 2019.

[19] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '08. New York, NY, USA: ACM, 2008, pp. 137–146.

[20] M. Paolieri, J. Mische, S. Metzlaff, M. Gerdes, E. Quiñones, S. Uhrig, T. Ungerer, and F. J. Cazorla, "A hard real-time capable multi-core smt processor," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 3, pp. 79:1–79:26, Apr. 2013.

[21] D. Bui, E. Lee, I. Liu, H. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in *Design Automation Conference (DAC)*, June 2011, pp. 274 – 279.

[22] D. Kirk and J. Strosnider, "Smart (strategic memory allocation for real-time) cache design using the mips r3000," in *Proc. of the 11th RTSS*, 1990, pp. 322–330.

[23] C. Liu, A. Sivasubramaniam, and M. Kandemir, "Organizing the last line of defense before hitting the memory wall for cmps," in *Proc. of the 10th HPCA*. USA: IEEE, 2004, pp. 176–.

[24] R. Iyer, "CQoS: A framework for enabling qos in shared caches of cmp platforms," in *Proc. of the 18th ICS*. USA: ACM, 2004, pp. 257–266.

[25] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural support for operating system-driven cmp cache management," in *Proc. of the 15th PACT*. USA: ACM, 2006, pp. 2–12.

[26] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proc. of the 45thDAC*. USA: ACM, 2008, pp. 300–303.

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

[28] D. Sanchez and C. Kozyrakis, "The zcache: Decoupling ways and associativity," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 187–198.

[29] ——, "Vantage: Scalable and efficient fine-grain cache partitioning," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 57–68.

[30] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, Dec 2006, pp. 423–432.

[31] A. Chousein and R. N. Mahapatra, "Fully associative cache partitioning with don't care bits for real-time applications," *SIGBED Rev.*, vol. 2, no. 2, pp. 35–38, Apr. 2005.

[32] A. Asaduzzaman, F. N. Sibai, and M. Rani, "Improving cache locking performance of modern embedded systems via the addition of a miss table at the {L2} cache level," *Journal of Systems Architecture*, vol. 56, no. 4-6, pp. 151–162, 2010.

[33] A. Sarkar, F. Mueller, and H. Ramaprasad, "Predictable task migration for locked caches in multi-core systems," in *Proc. of the LCTES'11*. New York: ACM, 2011, pp. 131–140.

[34] J. Reineke, I. Liu, H. Patel, S. Kim, and E. Lee, "PRET DRAM controller: Bank privatization for predictability and temporal isolation," in *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2011.

[35] Z. Wu, Y. Krish, and R. Pellizzoni, "Worst Case Analysis of DRAM Latency in Multi-Requestor Systems," in *Real-Time Systems Symposium (RTSS)*, 2013.

[36] M. Paolieri, E. Quiñones, J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time CMPs," *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, 2009.

[37] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *Hardware/software codesign and system synthesis (CODES+ISSS)*. ACM, 2007.

[38] Y. Zhou and D. Wentzlaff, "Mitts: Memory inter-arrival time traffic shaping," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 532–544.

[39] P. K. Valsan and H. Yun, "Medusa: A predictable and high-performance dram controller for multicore based embedded systems," in *Cyber-Physical Systems, Networks, and Applications (CPSNA), 2015 IEEE 3rd International Conference on*, Aug 2015, pp. 86–93.

[40] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2011, pp. 374–385.

[41] F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun, "Deterministic Memory Abstraction and Supporting Multicore System Architecture," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 1:1–1:25.

[42] B. Li, L. Zhao, R. Iyer, L.-S. Peh, M. Leddige, M. Espig, S. E. Lee, and D. Newell, "Coqos: Coordinating qos-aware shared resources in noc-based socs," *J. Parallel Distrib. Comput.*, vol. 71, no. 5, pp.

700–713, May 2011. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2010.10.013

[43] N. Rafique, W. Lim, and M. Thottethodi, "Architectural support for operating system-driven cmp cache management," in *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2006, pp. 2–12.

[44] A. Sharifi, S. Srikantaiah, A. Mishra, M. T. Kandemir, and C. R. Das, "Mete: Meeting end-to-end qos in multicores through system-wide resource management," vol. 39, 01 2011, pp. 13–24.

[45] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses, "Rate-based qos techniques for cache/memory in cmp platforms," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 479–488. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542342

[46] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "Qos policies and architecture for cache/memory in cmp platforms," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 25–36, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1269899.1254886

[47] B. Li, L.-S. Peh, L. Zhao, and R. Iyer, "Dynamic qos management for chip multiprocessors," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 3, pp. 17:1–17:29, Oct. 2012. [Online]. Available: http://doi.acm.org/10.1145/2355585.2355590

[48] J. Ma, X. Sui, N. Sun, Y. Li, Z. Yu, B. Huang, T. Xu, Z. Yao, Y. Chen, H. Wang, L. Zhang, and Y. Bao, "Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard)," *SIGPLAN Not.*, vol. 50, no. 4, pp. 131–143, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2775054.2694382

[49] A. Wolfe, "Software-based cache partitioning for real-time applications," *Journal of Computing Software Engineering*, vol. 2, no. 3, pp. 315–327, Mar. 1994.

[50] F. Mueller, "Compiler support for software-based cache partitioning," in *Proc. of the ACM SIGPLAN LCTES*. USA: ACM, 1995, pp. 125–133.

[51] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. of the EMSOFT'09*. ACM, 2009, pp. 245–254.

[52] B. Ward, J. Herman, C. Kenna, and J. Anderson, "Making shared caches more predictable on multicore platforms," in *Proc. of the 25th ECRTS*, 2013, pp. 157–167.

[53] G. Gracioli and A. A. Fröhlich, "An experimental evaluation of the cache partitioning impact on multicore real-time schedulers," in *Proc. of the 19th IEEE RTCAS*. IEEE, 2013.

[54] J. Liedtke, H. Haertig, and M. Hohmuth, "Os-controlled cache predictability for real-time systems," in *Proc. of the 3rd IEEE RTAS*. USA: IEEE, 1997, pp. 213–224.

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

[56] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proc. of the 5th IEEE/ACM CODES+ISSS*. USA: ACM, 2007, pp. 143–148.

[57] H. Ding, Y. Liang, and T. Mitra, "Wcet-centric partial instruction cache locking," in *Proc. of the 49th ACM/IEEE DAC*, June 2012, pp. 412–420.

[58] A. Arnaud and I. Puaut, "Dynamic instruction cache locking in hard real-time systems," in *Proc. of the 14th RTNS*, 2006.

[59] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proc. of ACM SIGMETRICS*. New York, NY, USA: ACM, 2003, pp. 272–282.

[60] T. Liu, M. Li, and C. Xue, "Instruction cache locking for real-time embedded systems with multi-tasks," in *Proc. of the 15th IEEE RTCSA*, Aug 2009, pp. 494–499.

[61] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals, "Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 695–706, 2011.

[62] H. Ding, Y. Liang, and T. Mitra, "Wcet-centric dynamic instruction cache locking," in *Proc. of DATE*, March 2014, pp. 1–6.

[63] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "Rtos support for multicore mixed-criticality systems," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, April 2012, pp. 197–208.

[64] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, "WCET(m) estimation in multi-core systems using single core equivalence," in *Proc. of the 27th ECRTS*, July 2015.

[65] L. Sha, M. Caccamo, R. Mancuso, J. E. Kim, M. K. Yoon, R. Pellizzoni, H. Yun, R. B. Kegley, D. R. Perlman, G. Arundale, and R. Bradford, "Real-time computing on multicore processors," *Computer*, vol. 49, no. 9, pp. 69–77, Sept 2016.

[66] M. Xu, L. T. X. Phan, H. Choi, Y. Lin, H. Li, C. Lu, and I. Lee, "Holistic resource allocation for multicore real-time systems," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2019, pp. 345–356.

[67] R. Tabish, R. Mancuso, S. Wasly, A. Alhammad, S. S. Phatak, R. Pellizzoni, and M. Caccamo, "A real-time scratchpad-centric OS for multicore embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016 IEEE 22th*, April 2016.

[68] S. Wasly and R. Pellizzoni, "A dynamic scratchpad memory unit for predictable real-time embedded systems," in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*. IEEE, 2013, pp. 183–192.

[69] M. R. Soliman and R. Pellizzoni, "WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 76. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 24:1–24:23.

[70] J. Freitag, S. Uhrig, and T. Ungerer, "Virtual Timing Isolation for Mixed-Criticality Systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Altmeyer, Ed., vol. 106. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 13:1–13:23. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/8990

[71] A. K. Jain, S. Lloyd, and M. Gokhale, "Microscope on memory: Mpsoc-enabled computer memory system assessments," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2018, pp. 173–180.