Arm DynamIQ Shared Unit and Real-Time: An Empirical Evaluation

Ashutosh Pradhan, Daniele Ottaviano, Yi Jiang, Haozheng Huang, Alexander Zuepke, Andrea Bastoni, Marco Caccamo
Technical University of Munich
Email: {ashutosh.pradhan, daniele.ottaviano, yi9.jiang, haozheng.huang, alex.zuepke, andrea.bastoni, mcaccamo}@tum.de

***Abstract*—The increasing complexity of embedded hardware platforms poses significant challenges for real-time workloads. Architectural features such as Intel RDT, Arm QoS, and Arm MPAM are either unavailable on commercial embedded platforms or designed primarily for server environments optimized for average-case performance and might fail to deliver the expected real-time guarantees. Arm DynamIQ Shared Unit (DSU) includes isolation features—among others, hardware per-way cache partitioning—that can improve the real-time guarantees of complex embedded multicore systems and facilitate real-time analysis. However, the DSU also targets average cases, and its real-time capabilities have not yet been evaluated.**

This paper presents the first comprehensive analysis of three real-world deployments of the Arm DSU on Rockchip RK3568, Rockchip RK3588, and NVIDIA Orin platforms. We integrate support for the DSU at the operating system and hypervisor level and conduct a large-scale evaluation using both synthetic and real-world benchmarks with varying types and intensities of interference. Our results make extensive use of performance counters and indicate that, although effective, the quality of partitioning and isolation provided by the DSU depends on the type and the intensity of the interfering workloads. In addition, we uncover and analyze in detail the correlation between benchmarks and different types and intensities of interference.

***Index Terms*—real-time system, multi-core, cache partitioning**

I. Introduction

The complexity of embedded real-time systems is growing exponentially. Powerful AI-driven algorithms are increasingly being deployed at the edge, driven by the growing data demands of advanced sensors. Embedded hardware platforms are evolving to meet these trends by delivering enhanced computational capabilities while minimizing energy consumption.

These platforms not only integrate dedicated accelerator units (*e.g.,* TPUs or DSPs) but also combine multiple types of cores with different computational and energy profiles [1], [2]. Notably, to facilitate communication and optimize data flows, cores and accelerators share the same memory hierarchy.

From a real-time perspective, the increasing complexity of such hardware platforms introduce significant challenges to system analysis. Over the years, the real-time systems community has proposed numerous techniques at both the software and hardware levels to address these challenges, aiming to reduce complexity and enhance the overall predictability of embedded multicore systems. The shared memory hierarchy has received particular attention, with prior research extensively focusing on cache, memory, and interconnect subsystems (*e.g.,* [3]–[6]).

On the architectural side, technologies like Intel RDT [7], Arm QoS [8], and Arm MPAM [9] enable different levels of quality of service for software layers. The real-time characteristics of RDT and Arm QoS have been exploited in previous studies (*e.g.,* [10]–[12]). However, these techniques generally target average-case scenarios, and corner cases may undermine the predictability characteristics they can offer to real-time applications [13]. Furthermore, the lack of processors implementing MPAM prevents its practical evaluation [14].

As an evolution of Arm's big.LITTLE architecture, DynamIQ technology combines performance and energy-efficient cores into fully integrated clusters. The DynamIQ Shared Unit (DSU) [15] provides a per-cluster last-level cache (L3) and includes logic for interconnecting multiple DynamIQ clusters. The DSU is adopted in most of the new Arm v8 and v9 designs. For instance, the recent Rockchip RK3568 and RK3588, as well as the NVIDIA Orin AGX [16]–[18], incorporate up to three DSUs to connect single or multiple clusters of Cortex-A55, A76, or A78 cores. For real-time applications, the DSU is particularly interesting as it introduces, among other features, the ability to partition the L3 cache into up to four way-based partitions.

Similar to Intel RDT [10], it is foreseeable that the DSU's partitioning capabilities will become common in real-time systems community. However, the real-time capabilities of the DSU require close scrutiny. For example, is its hardware-based way partitioning to be preferred to software-based set partitioning (*cache coloring*)? Or, what are the corner cases and limits of the DSU to support real-time requirements?

This paper provides the first comprehensive evaluation of Arm DSU capabilities for real-time workloads. Specifically, we present a large-scale evaluation of three different deployments of the DSU on Rockchip RK3568, RK3588, and NVIDIA Orin, as well as a comparison with the well-studied AMD Zynq UltraScale+ ZCU102 [19] (Sec. III shows details and differences among these boards). Using both synthetic and real-world benchmarks from the *San Diego Vision Benchmark Suite* (*SD-VBS*) [20], we investigate the partitioning capabilities of the DSU when applications are subject to different types and intensities of interference. Furthermore, using the Jailhouse hypervisor (HV) [21], which has proven effective in prior studies [11], [12], [22], we evaluate and compare

way and set partitioning strategies on DSU-based boards in such environments. On the ZCU102, we further underline differences with color-based partitioning, which is the only viable cache-isolation mechanism on previous generations of Arm chips. In all evaluations, we use carefully selected performance counters to provide detailed insights.

Our results show a complex picture. Although DSU-based partitioning is effective in protecting real-time workloads from interference, the quality of isolation strongly depends on the interfering workload and its intensity. Our findings also indicate that platform-specific optimizations, such as *prefetching* or *write-streaming*, can considerably impact the results. Additionally, to the best of our knowledge, this paper is the first to provide in-depth details on the correlations between different *SD-VBS* benchmarks and both types and intensity of the interference.

In summary, the paper makes the following contributions:

- Details how to support DSU features on three different boards for both Linux- and hypervisor-based setups.
- Uses both synthetic and real-world benchmarks to evaluate the real-time capabilities of DSU for different types and intensities of interference.
- Compares way and set partitioning techniques and highlights differences to setups for previous core generations.
- Exploits the rich performance counter architecture to provide in-depth insights and unveil corner case behaviors on different platforms.

In the following: Sec. II discusses previous works. Sec. III presents background on the memory subsystem for Arm embedded-boards. Sec. IV and Sec. V detail the experimental setup, experiments, and observations. Sec. VI concludes.

II. Related Work

The issue of shared resource management in multicore architectures has been well-studied in the context of real-time systems. Uncontrolled interference from co-running cores introduces non-determinism in the computation of response time and worst-case execution time (WCET) analysis. Previously, attempts have been made to define a single-core equivalent of a multicore commercial off-the-shelf (COTS) architecture [23], [24]. To define such a single-core equivalent, one must identify and account for the major sources of interference on COTS multicore architectures—namely, the shared caches, the shared interconnect, and the shared main memory [25].

To improve the predictability in accessing shared caches, diverse cache management techniques have been proposed, which can be mostly aggregated into *cache-partitioning techniques* and *cache-locking techniques* [25], [26]. Cache partitioning divides the cache into smaller chunks that can be assigned to specific tasks or cores. In a set-associative cache architecture, the cache can be divided into specific ways (way partitioning) or specific sets (set partitioning). Generally, the methods proposed for way partitioning require modifications of the cache implementation in hardware [27]–[30]. The DSU implements way partitioning at the hardware level. Set partitioning, on the other hand, has been implemented leveraging hardware features [31]–[33] or purely in software [22], [34]–[37]. A popular method for implementing set partitioning in software is *page coloring*, which can be implemented by changing the page table management at the operating system (OS) level [3], [37], or at the hypervisor level [22], [38]. Contrary to cache partitioning, cache locking pins certain cachelines to make the cache content and WCET more predictable [26], [39]. However, it often requires hardware support which is not readily available on current-generation COTS platforms [39]. For example, earlier 32-bit Arm v7-A architectures with at most L2 caches supported *cache lockdown* and *Lockdown-by-Master* (in the L2C-310 controller [40]). This feature was phased out in the transition from Cortex-A9 to Cortex-A15. The numerous differences between Arm v7-A CPUs that supported cache lockdown and those with DSU make systematic evaluations and direct comparisons with our work impractical. A combination of locking and partitioning can be used in synchronization to obtain higher levels of predictability [3].

Apart from shared caches, interference from competing cores in the shared interconnect and main memory can affect real-time performance. Most COTS platforms employ Dynamic Random Access Memory (DRAM) as the main memory, which is ill-suited to provide real-time guarantees [41]. Memory access requests from non-real-time tasks can complicate the ability to provide a tight WCET for real-time tasks. To improve the predictability of memory accesses, various works propose changes to the existing hardware [41]–[43]. Notably, the DRAM controller has been redesigned to provide an analytical bound on the maximum latency for each memory request [44]–[46]. In contrast to the methods that require hardware changes, several works rely on making OS-level software modifications to allow deterministic main memory access. For example, in [5], the authors propose PALLOC, a memory allocator in OS that reduces inter-core memory interference by partitioning the DRAM banks. Approaches like MemGuard [4] and MemPol [47] propose a per-core memory bandwidth regulation system using performance counters. Along similar lines, Sohal *et al.* [11] propose a framework for profiling the temporal workload on both CPU and accelerators using a similar approach to memory bandwidth regulation.

In recent years, new features have been introduced in COTS platforms that can aid real-time execution of applications. In 2015, Intel introduced the Resource Director Technology (RDT) to monitor shared platform resources [7]. Similarly, QoS features exist at the interconnect level in Arm and AMD [8], [48]. Arm MPAM [9] holds promise from the real-time perspective, but it is yet to be implemented in COTS platforms. Xu *et al.* [10] use the Cache Allocation Technology (CAT) feature of Intel RDT to provide an algorithm for partitioned scheduling of tasks in a multicore environment. However, the real-time features in RDT, such as cache partitioning, are not always effective in practice [13]. The recently introduced Arm DSU offers cache-partitioning of the shared cache in a cluster. However, a detailed study of the feature and its real-time applications has yet to be done.

III. Background

A. General

We present a brief overview of the cache architecture on current 64-bit Arm platforms. The Arm cores[1] discussed in this paper use a cacheline size of 64 B at all levels of their memory hierarchy. Therefore, the lowest 6 bits of an address `addr[5:0]` denote a specific *byte offset* in a cacheline. The next higher $\log_2 n_{sets}$ bits of an address, *e.g.,* `addr[15:6]` for 1024 sets in the L2 cache on the ZCU102, define the *set index* into a specific cache (see Fig. 1a). Additionally, the set index can include information of further address bits, as in the case of the XOR permutation on the Cortex-A78; see Sec. III-H.

B. Cache Set Partitioning

In a cache, if the address range covered by set index and offset is larger than the page size, *page coloring* can be used, as Fig. 1a shows: The physical address bits beyond the page size defines the specific page color, *e.g.,* 16 colors (4 bits) for a page size of 4 KB (`addr[11:0]`) in the L2 cache on the ZCU102, while the virtual address bits ensure a contiguous mapping of the physical memory to the application, set by the OS [3] or the hypervisor [38].

C. Cache Way Partitioning

In contrast, way partitioning reserves a specific number of ways for a partition. In the Arm DSU, a group of four ways can be assigned to one or more *DSU scheme IDs* (up to eight). Each core can then be assigned a scheme ID as Fig. 1b shows. The L3 cache in the DSU has 16 ways, so effectively, four partitions are available to the eight cores supported by a DSU [15]. On Arm, way partitioning effectively selects cache ways in the L3 cache eligible to *allocate* data to when data is *evicted* from the higher level L1 or L2 caches. Therefore, applications can always access all cachelines in the cache, unlike in the MMU-based set partitioning. As way groups can be assigned to multiple partitions, arbitrary combinations of data sharing are possible.

D. Memory Accesses and Architectural Optimizations

Modern architectures optimize memory accesses through various mechanisms that influence cache behavior. Memory operations generally fall into three categories: *reads*, which load data from memory into registers; *writes*, which store data from registers back to memory; and *modifications*, which involve read-modify-write sequences such as atomic operations.

To improve performance, recent Arm CPUs employ *hardware prefetching*, which anticipates future memory accesses and loads data into caches before it is explicitly requested. While this reduces memory latency, it introduces additional memory operations. Additionally, to optimize memory bandwidth for large data transfer, the *write-streaming* mechanism enables writes to bypass the L3 cache entirely. This can introduce latency inconsistencies depending on memory bandwidth contention. Prefetching and write-streaming are unavoidable and activated automatically in Arm systems today, and even simple operations like `memcpy` or `memset` can trigger them. Understanding their impact on cache partitioning is necessary for accurately assessing system behavior and ensuring performance isolation.

[1]In the paper, we use the terms *core* and *CPU* interchangeably.

E. Cortex-A53

Introduced in 2012, the Cortex-A53 is a power-efficient in-order design in the first generation of 64-bit Arm cores [49]. The Cortex-A53 supports a private 2-way set-associative L1 instruction cache and a private 4-way set-associative L1 data cache, both with configurable sizes between 8 KB to 64 KB. According to the documentation [49], both L1 caches use a *pseudo-random cache replacement policy*. The Cortex-A53 predates the DSU, and thus cannot be used together with it, however, cores can be instantiated in clusters of up to four cores that share the L2 memory subsystem. For more cores, multiple clusters of the same or different core types must be instantiated. All cores in a cluster share the same optional 16-way set-associative L2 cache with sizes from 128 KB to 2 MB. The shared L2 cache is used as a *victim cache* and is *exclusive* to the L1 caches, *i.e.,* data is not allocated in the L2 cache on misses in L1. Cachelines get allocated in the L2 cache only when they are evicted from the cores' L1 caches. Arm documents the internal format of the tag RAMs of TLBs and L1 caches, but not of the L2 cache [49].

F. Cortex-A55

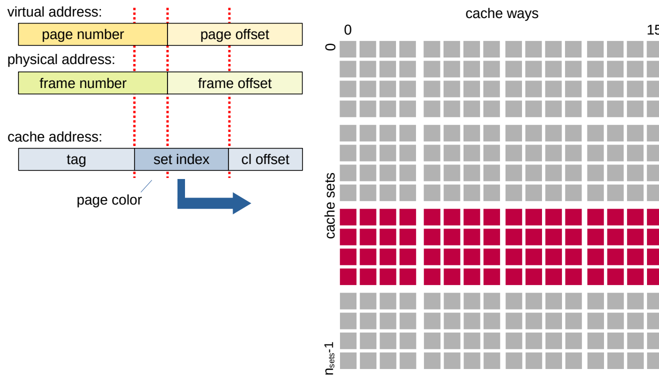
The Cortex-A55 is the power-efficient successor of the Cortex-A53 for DSU-based designs [50]. Like its predecessor, it uses an in-order pipeline design. It supports 4-way set-associative instruction and data L1 caches with sizes between 16 KB and 64 KB. The *age* bits in the documentation of the tag RAMs indicate LRU or pseudo-LRU (PLRU) as replacement policy for the L1 data cache. The optional 16-way set-associative L2 cache is private to the core with sizes between 64 KB and 256 KB. Like on the Cortex-A53, the L2 cache is a *victim cache* and is *exclusive* to the L1 data cache.

G. Cortex-A76

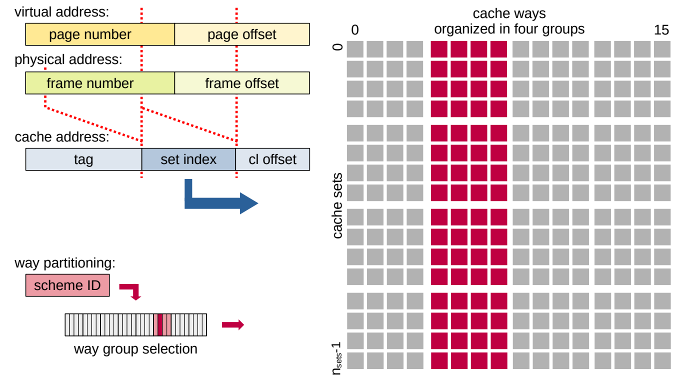
The Cortex-A76 is the second generation of out-of-order performance cores for DSU-based designs [51] and typically paired with Cortex-A55 cores. The Cortex-A76 has 4-way set-associative L1 data and instruction caches of 64 KB size. Arm documents a PLRU cache replacement policy for both. The 8-way set-associative L2 cache is of 128 KB to 512 KB size. The L2 cache is mandatory and *inclusive* to the L1 data cache, *i.e.,* any data allocated in the L1 cache also takes space in the L2 cache. The documentation mentions a *dynamic biased replacement policy* with seven *PLRU* bits in the L2 victim RAM [51]. The Cortex-A76 is also available in a Cortex-A76AE variant for safety-critical applications [52].

H. Cortex-A78

The Cortex-A78 is the fourth generation of out-of-order performance cores for DSU-based designs [53] and can also be



(a) Cache set partitioning using page coloring.



(b) Cache way partitioning in Arm DSU.

Fig. 1: Cache tag and set index are derived from the physical address. The set index selects a set (row) in the cache. For a cache hit, the tag matches a corresponding tag in one of the ways (column). (a) Set partitioning: bits beyond the page size in the set index select the cache color, *i.e.,* a specific range of sets. The set index requires a linear dependency on the address. (b) Way partitioning: the ID register selects one of eight schemes in the group selection register. The four bits indicate the group of ways private to the scheme. The set index can be a non-linear function, *e.g.,* permutated from upper and lower address bits.

paired with power-efficient Cortex-A55 cores. The Cortex-A78 uses 4-way set-associative L1 data and instruction caches of 32 KB or 64 KB size. Arm documents a PLRU cache replacement policy for both L1 caches. The 8-way set-associative L2 cache is 256 KB or 512 KB size. As in the Cortex-A76, the L2 cache is inclusive with the L1 cache. Arm lists 24 bits in the L2 victim RAM for the replacement policy, but differently from the other cores, the L2 cache uses a permutation function to derive the set index, *e.g.,* `addr[22:15] XOR addr[14:7]` for the 256 KB configuration [53], as shown in Fig. 1b. A Cortex-A78AE variant is also available [54].

I. DSU

Announced in 2017 with the Cortex-A55 and A75 cores, the *DynamIQ Shared Unit* (DSU) interfaces the cores and provides the L3 memory system in an SoC [15]. The DSU supports up to eight cores in up to three internal clusters of *big*, *little* and *other* cores of the same microarchitecture. To support more than eight cores, multiple instances of the DSU must be instantiated. The DSU-AE variant additionally implements lock-step functionality for pairs of two AE cores [55]. The DSU provides an optional 16-way set-associative shared L3 cache with sizes from 256 KB to 4096 KB. The L3 cache can be implemented physically in one or two slices (memory blocks). The address bit to select the slice is configurable. Power management allows *all*, $\frac{3}{4}$, $\frac{1}{2}$, $\frac{1}{4}$ or *none* of the cache to be powered down. For cache sizes of 1536 KB and 3072 KB, the number of available ways in the cache is reduced to 12, and $\frac{1}{4}$ of the cache is permanently powered down. The DSU implements *cache-way partitioning* in the L3 cache with a granularity of groups of four ways, as described in Sec. III-C. When cores like the Cortex-A55 are configured without L2 caches, the shared L3 cache appears to them as L2 cache. However, adhering to Arm documentation, we indicate the DSU cache as L3 cache and denote the L2 cache as missing. The DSU provides its own Performance Monitoring Unit (PMU). The `CLUSTERPM` PMU registers are shared between the cores and monitor cache and bus activity of all cores.

IV. Setup

Our in-depth, exhaustive experiments focus on isolation and real-time properties. To this end, we compare and analyze two implementations of cache-partitioning: set partitioning, enabled through Jailhouse's cache coloring [56], and way partitioning, achieved through a custom driver controlling the Arm DSU cache partitioning feature.

A. SoCs, Boards, and Configuration

We refer to the specific boards by the name of their SoCs, as the ability to use cache partitioning is a property of the SoC and the firmware. Boards using the same SoC might have different configurations, *e.g.,* use DDR4 vs LP-DDR4, but this does not change the general observations.

ZCU102: The ZCU102 is a reference board for the AMD Zynq UltraScale+ SoC family, which supports four Cortex-A53 cores with 32 KB private L1 caches and a shared L2 cache of 1 MB size [19]. We use Linux kernel 6.1_LTS of the Linux repository from Xilinx [57]. We note that this board lacks the DSU and is primarily included as a baseline platform for comparison, given its frequent use in prior research on cache partitioning and specifically on set partitioning [58], [59].

RK3568: The Rockchip RK3568 SoC supports four Cortex-A55 cores with 32 KB private L1 caches and a shared LLC of 512 KB size in the DSU [16]. This SoC lacks private L2 caches for the cores and, as previously noted, we refer to the DSU cache as L3 cache and denote an L2 cache size of zero for consistency with the other SoCs. We evaluate the SoC on a Youyeetoo YY3568 board using Debian 12 and a mainline Linux kernel 6.1.

RK3588: The Rockchip RK3588 SoC supports four Cortex-A55 cores with 32 KB private L1 caches and 128 KB private L2 caches, four Cortex-A76 cores with 64 KB private L1

caches and 512 KB private L2 caches, and a shared L3 cache of 3 MB size in the DSU [17]. We performed our experiments on both the Orange Pi 5 Plus and Raxda ROCK 5B boards. The results are similar, and we consider only the Orange Pi 5 Plus for the rest of this paper. We again use Debian 12 and a mainline Linux kernel 6.11-rc4.[2]

Orin: We evaluate the NVIDIA Orin SoC on the NVIDIA Orin AGX Devkit. In this configuration, the Orin provides three DSU-AE clusters of four Cortex-A78AE each. The cores are configured with 64 KB private L1 caches, 256 KB private L2 caches, 2 MB L3 caches per DSU, and a 16-way set-associative L4 cache of 4 MB size. We use Jetson Linux 36.4 (based on Ubuntu 22.04) with Linux kernel 5.15 and the default power mode (30W, mode ID 2). We note that extending Jailhouse support to the Orin is a non-trivial effort that is still ongoing. Therefore, we defer the analysis of set partitioning for the NVIDIA Orin to future work.

B. Evaluation Methodology

For each hardware setup, we evaluate the real-time performance of the two cache-partitioning implementations by assessing the temporal behavior of *SD-VBS* [20], [60] running alongside a memory-intensive synthetic interference application on separate cores, both described in detail in Sec. IV-D and Sec. IV-E. The evaluation is conducted by varying two factors: the cache allocation between the benchmark and the interference, and the work-set size of the interference task.

Specifically, each benchmark is executed first without any cache partition. Then, the cache is partitioned using both way and set techniques (as applicable) by allocating four cache partitions between the benchmark and interference applications in three ratios:

- **1/3**: $\frac{1}{4}$ to the benchmark and $\frac{3}{4}$ to the interference
- **2/2**: $\frac{1}{2}$ to the benchmark and $\frac{1}{2}$ to the interference
- **3/1**: $\frac{3}{4}$ to the benchmark and $\frac{1}{4}$ to the interference

All configurations are tested by varying the working-set size (*"intensity"*) of the interference from 8 KB to 8 MB. We evaluate the real-time performance of partitioning when the interference impacts different layers of the memory hierarchy, from the L1 cache to the DRAM. The minimum working set size is smaller than the L1 cache and the maximum is larger than two times the size of the LLC for each platform.

1) Set Partitioning Setup: We rely on the cache-coloring implementation in the Minerva fork of Jailhouse [56] for the *ZCU102*, as it provides fine-grained control over cache set partitioning (*e.g.,* [22]). We extended Jailhouse to support our Rockchip platforms (both Cortex-A55 and A76 cores), integrating support for cache coloring. The work has been upstreamed [56]. To have a fair comparison between set and way partitioning, we avoid choosing configurations that would color the L1 cache of a core, and we carefully select configurations that—like on way partitioning—only partition the LLC.

[2]The vendor kernel 5.15 is tuned for aggressive power saving and Android use cases and produces unstable benchmark results.

2) Way Partitioning Setup: We develop an ad-hoc *driver* (specifically, an OS driver and a HV-extension) to control the DSU registers at both Linux and Jailhouse levels. Using the driver, we (I) initially assign a unique ID $0..n-1$ to each core's `CLUSTERTHREADSID` registers, and then we (II) assign the way groups in the `CLUSTERPARTCR` register.

However, by default, access to the DSU PMU and cluster configuration registers is disabled at the hypervisor and operating system level. Therefore, we also modify the Arm Trusted Firmware (ATF) and unconditionally enable the `SMEN`, `TSIDEN`, and `CLUSTERPMUEN` bits in both `ACTLR_EL3` and `ACTLR_EL2` registers on all available cores. This allows us to configure DSU cache partitioning at Linux kernel level for our experiments. Note that hypervisors should not configure `ACTLR_EL2` to disable access to the cache configuration registers at OS level in a VM.

C. Performance Counters in Cortex-A55, A76, A78 and DSU

Nearly one hundred PMU events are available per Arm core, and over forty within the DSU. Still, only a subset can be counted at any given time due to the limited number of Performance Monitoring Counters (PMCs) available on both the cores and DSU. We use the following per-core and DSU events to analyze cache and memory interference.

1) Arm Cores Events: `L1D_CACHE_REFILL_INNER (0x44)` and `L1D_CACHE_REFILL_OUTER (0x45)` measure L1 data cache refills, distinguishing between refills sourced from within the cluster (like the L3 cache or from another core) versus those from external sources. This distinction aids in assessing the locality of memory access and the pressure placed on the L1 cache by external memory transactions. The `L3D_WS_MODE (0xc7)` event (only available on Cortex-A55) records cycles in which the core operates in *write-streaming* mode, avoiding L3 allocations. This allows for assessing the impact of sustained write operations on cache utilization and memory coherence.

2) DSU Events: `SCU_PFTCH_CPU_ACCESS (0x500)` tracks L3 prefetch transactions initiated by the CPU, giving insight into the frequency and distribution of L3 cache prefetching. Complementing this, `SCU_PFTCH_CPU_HIT (0x502)` and `SCU_PFTCH_CPU_MISS (0x501)` count prefetch hits and misses, respectively, helping to measure prefetching effectiveness in reducing memory latency. Additionally, `L3D_ACCESS (0x2b)` is a comprehensive counter that captures all read and write transactions directed to the DSU, providing an aggregated view of L3 activity. `BUS_ACCESS (0x19)` monitors transactions between the DSU and the interconnect, indicating the volume of external memory traffic used to understand memory transaction demands. Finally, `L3D_CACHE_ALLOCATE (0x29)` counts L3 cache allocations that do not involve refills. We use it to observe how cache space is utilized without triggering additional memory latency.

D. Dataset – RT-Bench Vision

This setup uses the RT-Bench framework [60], which wraps the *SD-VBS* suite to set up real-time behavior: RT-Bench

enables periodic task execution, the use of real-time task priorities, and precise temporal measurements such as latency and execution time.[3]

The *San Diego Vision Benchmark Suite* used for the assessment includes the following applications:

- *disparity* computes depth information by identifying pixel correspondences between stereo images.
- *sift* (Scale-Invariant Feature Transform) detects and matches distinctive features in images, providing robustness across scale and orientation changes.
- *localization* assesses spatial positioning within an environment by leveraging visual landmarks.
- *mser* (Maximally Stable Extremal Regions) identifies stable regions across varying thresholds.
- *tracking* monitors the position of objects over time in video sequences, enabling applications in surveillance and interactive environments.

To improve the quality of the evaluation, we decided to use two datasets with different image resolutions, namely *cif* (352×288) and *vga* (640×480). The different datasets help understand the impact of computational-intensive workloads on real-time performances while partitioning the cache.

E. Interference-Bench

To obtain deep insights from experiments on *SD-VBS*, we focus on analyzing the effects of interfering workloads with varying working set sizes under different cache partitioning configurations. To achieve this, we developed a synthetic interference program (henceforth called interference-bench) which enables the generation of four types of interference workloads—*read*, *write*, *modify*, and *prefetch*—each with configurable working set sizes.

- *read* refers to reading of data from memory using a *load* instruction of a single byte. This requires the core to load the full cacheline to the L1 cache.
- *write* refers to writing of a cacheline to memory using *store* instructions to the whole cacheline.
- *modify* changes a single byte of a cacheline using a *store* instruction. The core thus needs to load the full cacheline and then apply the modification to it.
- *prefetch* refers to prefetching a cacheline for future use. In our experiments, we refer to prefetching to the L3 cache in RK3568, RK3588 and the Orin boards as prefetch.

We chose not to use other types of realistic applications as interference sources, as this synthetic interference benchmarks provides the most extreme memory usage patterns, revealing worst-case behaviors. This approach allows us to expose platform-specific corner cases that could arise in real-world scenarios under maximum stress conditions. Both the benchmarks and interference are implemented on Linux and the OS itself can be considered part of the working set.

[3]We use `dev/unstable` RT-Bench (https://gitlab.com/rt-bench/rt-bench/). It was cross-compiled on x86 Ubuntu 22.04 using aarch64-linux-gnu-gcc version 11.4.0

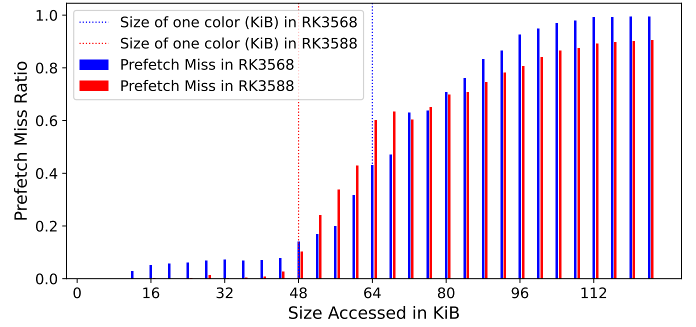


Fig. 2: Prefetch Miss Ratio vs the Size of memory accessed in the same color on the RK3568 and RK3588 platforms

Given the extensive number of experiments conducted, this paper presents only the most significant results that contribute directly to our observations in Sec. V.[4]

F. Bit Assignment for Set Partitioning

Set partitioning using *page coloring* techniques relies on the fact that the set index can be computed by masking specific (coloring) bits of a given physical address (see Sec. III). The presence of undocumented bit-scrambling and hash functions can result in losing control over the coloring bits and, consequently, in unexpected and erroneous behavior of the set partitioning implementation. Arm's DSU [15] does not document the presence or absence of bit-scrambling in set-index computation. To verify the correct behavior of set partitioning on the boards under test, we experimentally validate the following hypothesis: If we load a page of a given color as computed from the set index, it will replace the previous pages of the same color loaded into the cache. The experiment involves the following steps:

- Similar to [62], we use 2 MB Linux huge pages to target specific set-index in the cache. Since we need 21 bits to address a location inside a page of size 2 MB (the lower 21 bits of the virtual address), a memory location inside the page will be equal to the physical address.
- Using the huge page and software prefetch command, we sequentially load n 4 KB memory chunks of the same color to the L3 cache.
- We prefetch n 4 KB memory chunks of the same color to the L3 cache again and count the number of prefetch hits and misses using the PMCs (see Sec. IV-C).

We executed the experiment on both RK3568 and RK3588 using one Cortex-A55 core. Hardware prefetching was turned off by writing to the `L3PCTL` bit field of the `CPUECTRL` register to avoid interference from the hardware prefetcher. Following a similar calculation as in Sec. III-B, we obtain 8 colors on the RK3568 and 64 on the RK3588. Since the RK3568 has an L3 cache size of 512 KB, each color has a size of 64 KB. Similarly, each color on the RK3588 has a size of 48 KB.

In Fig. 2, we plot the miss ratio (number of misses observed divided by the maximum number of misses possible) for a

[4]The extended version of the paper with all results is available in [61].

given access size. We observe that the number of misses rises linearly as we cross the size of one color, indicating cache thrashing as content is loaded to the same previously occupied locations. This result, combined with the results from Obs. 1 and Obs. 2, strongly indicate that the DSU does not implement bit-scrambling in the color bits of the set index computation.

V. Experiments and Discussion

***Obs. 1:* Way partitioning can increase conflict misses**
Way partitioning effectively reduces the associativity of the cache. Therefore, access to memory that heavily relies on set associativity to achieve optimal performance can be penalized under way partitioning. For example, for workloads with strided access pattern (*i.e.,* that continuously write to the same sets), way-partitioned cache can have a higher execution time than a set-partitioned cache. To test this, we designed a synthetic benchmark for the RK3568 that simulates a workload that makes even use of the caches. The benchmark operates at three levels, using 1/4, 1/2, and 3/4 of the total sets available in the RK3568's L3 cache. Since the RK3568 has 512 sets, the levels correspond to 128, 256, and 384 sets, respectively. For each level, the benchmark linearly accesses the sets and modifies one cacheline $s \times w$ times, where s is the number of sets, and w is the number of ways—16 on the RK3568. We record the execution time of the benchmark under set and way partitioning and without partitioning. As shown in Fig. 3, the execution time for way partitioning is much higher than set partitioning for the same configuration due to the higher rate of conflict misses in the way-partitioned cache. Further, we observe that for 384 sets, the execution time under way-partitioning is slightly lower than for 256 sets. This hints at a recency bias in the replacement policy.

Despite the non-negligible effect that way partitioning can have on workloads that evenly access caches, in real environments, workloads are unlikely to present the strided access pattern used in our synthetic benchmark.

***Obs. 2:* Way and set partitioning are effective in reducing interference**
Following the methodology described in Sec. IV-B, we evaluate and visualize the behavior of cache-partitioning. Fig. 4 shows a subset of our results for both set and way partitioning using the *mser/vga* benchmark on the RK3568 (a–d) and RK3588 (e–h).[5] The figures report the slowdown ratio of the execution time of the benchmarks compared to the no-interference case for increasing size of the interference. Insets (a), (b), (e), (f) present results for *modify* interference, while (c), (d), (g), (h) for *prefetch* interference.

We observe that both way-partitioning and set partitioning significantly reduce the slowdown ratio compared to the case without partitioning. The effect of cache partitioning is more pronounced in the RK3588 boards than in the RK3568. In contrast to the synthetic example (see Obs. 1), the slowdown ratio in way partitioning is comparable to the slowdown ratio

[5]Other samples of the effects of cache-partitioning are shown in Figs. 8, 10, and 12. The full set of graphs is available in [61].

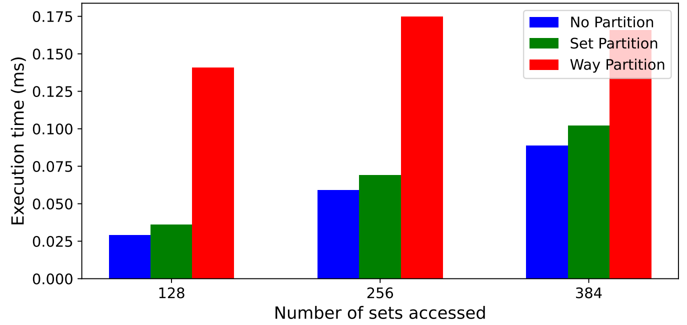


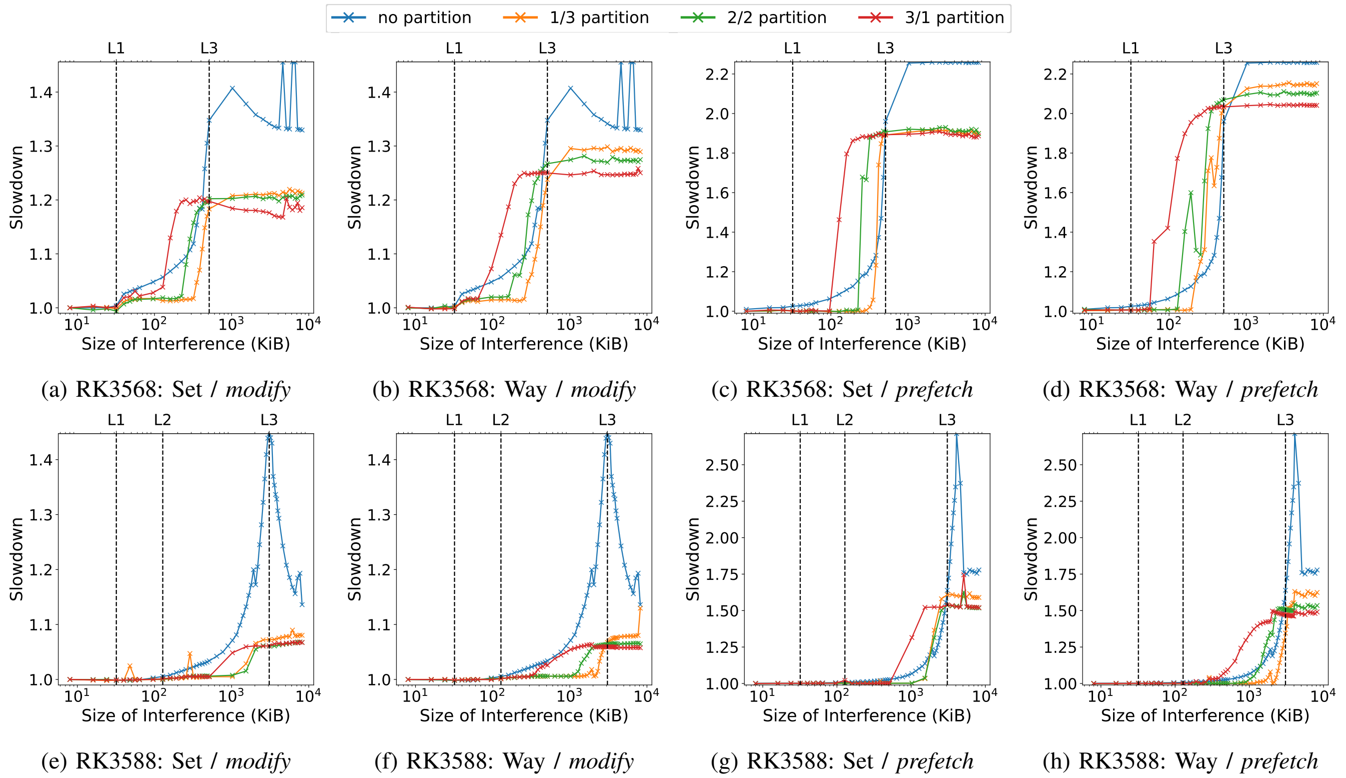
Fig. 3: Execution time of the synthetic benchmark under no partitioning, set partitioning, and way partitioning vs the number of sets accessed fully on the RK3568 platform

from set partitioning. In Fig. 4a and 4b, we can see that beyond the L3 cache size, set partitioning performs slightly better than way partitioning. On the other hand, in Fig. 4g and 4h, way partitioning performs slightly better than set partitioning.

***Obs. 3:* No partitioning method dominates the other**
Motivated by the results from Fig. 4, we carefully analyzed whether one of the two partitioning methods would consistently perform better than the other across several benchmarks and type/intensity of interference. Table I presents an example of our results for the maximum slowdown execution time ratio (relative to the no-interference case) for the *tracking* benchmark on the RK3568 and RK3588 platforms. For computing the execution times, the interference-bench is run on one of the A55 cores, and the benchmark is run on a different A55 core. We take the maximum of the slowdown ratio as the size of interference in the interference-bench varies from 8 KB to 8192 KB. Table I shows that the maximum slowdown ratio is similar between way and set partitioning across different interference patterns and datasets. In particular, for some scenarios, set partitioning performs better than way partitioning, while in other scenarios, the opposite holds true. We observed similar behaviors across the full set of benchmarks/interference types.

In summary, our results on practical benchmarks subject to interference indicate that no significant differences exist between the partitioning methods. Given the complexity of implementing cache coloring in software, way partitioning might be a more suitable choice for most of the real-world scenarios where no pathological access patterns (see Obs. 1) exist.

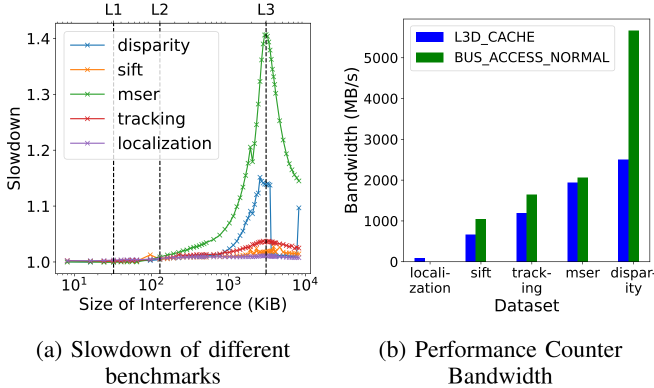
***Obs. 4:* Cache Partitioning affects performance only when the working-set size exceeds the private cache size**
If the working set of the benchmark fits in the private L1 and L2 caches, there is no interference from the other cores, and partitioning is not required. However, if the benchmark uses the shared LLC, we observe an expected slowdown depending on the interaction between the competing accesses to the LLC. Fig. 5a, reports the execution time slowdown (*w.r.t.* the no-interference case) of *localization*, *sift*, *tracking*, *mser*, and *disparity* (*vga* dataset) with interference from *modify* workload. The benchmarks and interference run on distinct A55 on the RK3588. Slowdown is negligible for *localization*



(a) RK3568: Set / *modify* (b) RK3568: Way / *modify* (c) RK3568: Set / *prefetch* (d) RK3568: Way / *prefetch*

(e) RK3588: Set / *modify* (f) RK3588: Way / *modify* (g) RK3588: Set / *prefetch* (h) RK3588: Way / *prefetch*

Fig. 4: Interference on *mser*/*vga* benchmark for RK3568 and RK3588 with *Set* and *Way* partitioning. Each plot reports the execution slowdown w.r.t the *no-interference* case with increasing size (KiB) of *modify* (a,b,e,f) and *prefetch* (c,d,g,h) interference. For each board, L1, L2 (if applicable), and L3 sizes are reported.



(a) Slowdown of different benchmarks

(b) Performance Counter Bandwidth

Fig. 5: Effect of *modify* interference on different benchmarks/*vga* dataset on the RK3588 platform without partitioning.

and *sift*, but is very noticeable for *disparity* and *mser* datasets.

To further investigate this behavior, in Fig. 5b, we plot the value of performance counter `L3D_CACHE` and `BUS_ACCESS`. The values provide information on the L3 cache accesses and main-memory accesses from the L3 cache (Sec. IV-C). The performance counters are normalized by multiplying for the size of the cacheline and dividing by the execution time of the benchmark (*i.e.,* converting to MB/s). We observe that the L3 access in *localization* is negligible, while *disparity* and *mser* have a high volume of transactions in the L3 cache and main memory. Therefore, when no longer executing on private caches, the effect of interference is high for *disparity* and *mser*, while it is negligible for *localization*. The rapid growth of *mser* slowdown in Fig. 5a is related to both the lower number of transactions in LLC and to the absolute execution time of *mser* and *disparity* (baselines: 0.19s and 0.71s, respectively). In fact, the absolute increase in execution time peaks at 0.10s in *disparity*, while it is less than 0.08s for *mser*.

***Obs. 5:* Accesses to the memory controller impact partitioning**
Fig. 4 shows that, despite limiting interference, both set and way partitioning still suffer from slowdowns. The slowdown starts increasing when the size of interference is less than the L3 cache and stabilizes beyond the L3 cache size. As visible in Fig. 6, such a slowdown is directly connected to an increase in the contention at the memory controller. Fig. 6 shows the `BUS_ACCESS` PMC for *modify* interference (used here as benchmark) on the RK3568 (units are normalized to MB/s as in Obs. 4). This reflects the pressure on the memory controller. Fig. 6 underlines that, without partitioning, the memory controller traffic increases around the size of the L3 cache. In fact, beyond the L3 cache size, *modify* starts accessing data that can no longer fit in the caches. Memory controller traffic therefore increases till it saturates to the maximum bandwidth.

However, under way partitioning, memory controller traffic increases much below the L3 cache size. This is because

TABLE I: Maximum execution time slowdown *w.r.t.* the no-interference case for the *tracking* benchmark on the RK3568 and RK3588 platforms for way/set partitioning and all interference types.

RK3568																	
		tracking															
		read				*write*				*modify*				*prefetch*			
	Partition Size	No	1/3	2/2	3/1	No	1/3	2/2	3/1	No	1/3	2/2	3/1	No	1/3	2/2	3/1
cif	way	1.05	**1.03**	1.03	1.03	1.33	**1.38**	**1.36**	1.35	1.03	1.02	1.02	1.02	1.13	**1.13**	**1.12**	**1.12**
	set		1.04	**1.03**	**1.03**		1.46	1.38	**1.33**		**1.02**	**1.02**	**1.01**		1.13	1.12	1.12
vga	way	1.28	1.17	**1.08**	1.09	1.59	2.8	**1.63**	**1.64**	1.14	1.09	**1.07**	**1.07**	1.44	1.55	1.34	1.34
	set		**1.15**	1.09	**1.07**		**2.74**	1.68	1.70		**1.07**	1.07	1.07		**1.48**	**1.34**	**1.32**
RK3588																	
		tracking															
		read				*write*				*modify*				*prefetch*			
	Partition Size	No	1/3	2/2	3/1	No	1/3	2/2	3/1	No	1/3	2/2	3/1	No	1/3	2/2	3/1
cif	way	1.04	**1.00**	1.01	**1.01**	1.03	**1.03**	**1.02**	**1.02**	1.05	**1.01**	1.01	**1.01**	1.10	**1.04**	**1.03**	**1.03**
	set		1.01	**1.01**	1.01		1.04	1.02	1.03		1.01	**1.01**	1.01		1.06	1.04	1.04
vga	way	1.03	**1.01**	1.01	**1.01**	1.04	**1.03**	1.04	1.05	1.03	**1.01**	1.01	**1.01**	1.10	**1.08**	**1.08**	**1.07**
	set		1.01	**1.01**	1.01		1.04	**1.04**	**1.05**		1.01	**1.01**	1.02		1.09	1.09	1.09

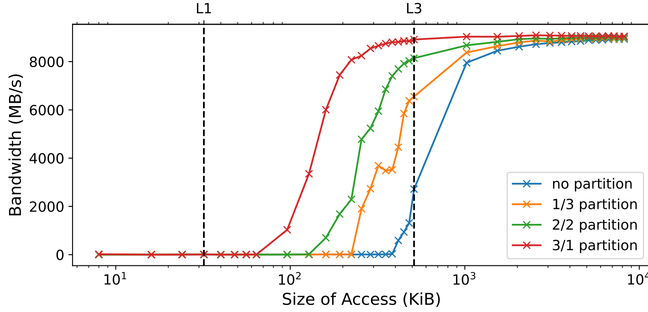


Fig. 6: Bandwidth of `BUS_ACCESS` as the size of *modify* access in interference-bench increases on the RK3568 platform

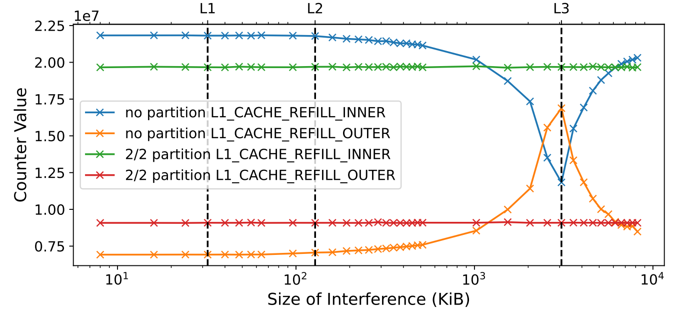


Fig. 7: Value of Performance Counters in single execution of the *mser/vga* benchmark in the presence of *modify* interference on RK3588 platform

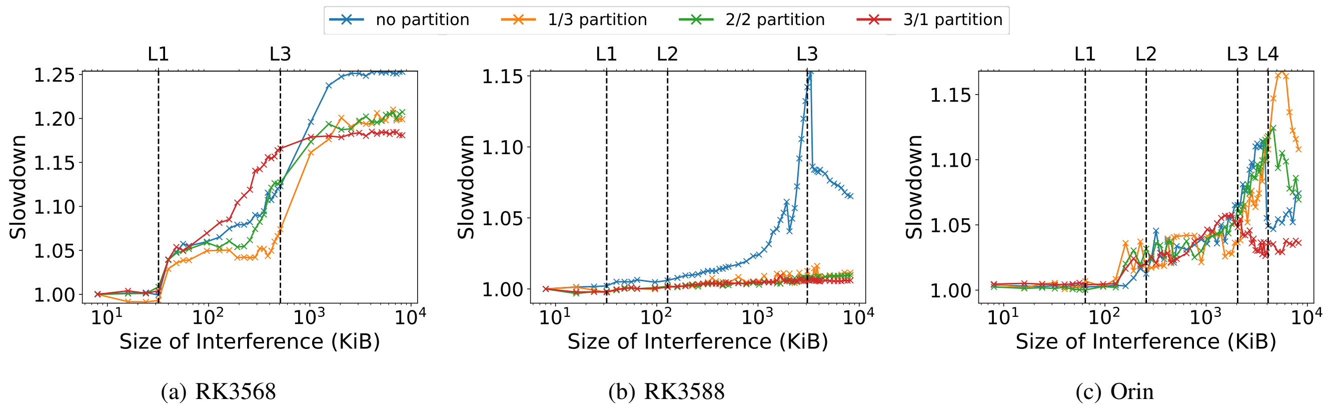
fewer LLCs is available to *modify*, hence early increasing the accesses to main memory. This point corresponds to the increase in slowdown for way partitioning as observed in Fig. 4b for each **1/3**, **2/2**, and **3/1** ratios. When the working set of the benchmarks does not fit in the L3 cache, the increased memory controller traffic from interference adversely affects their execution times.

***Obs. 6:* Reduced interference pressure causes a sharp speedup at the LLC size**

Most insets in Fig. 4, and *e.g.,* Fig. 5a and Fig. 8c present a slowdown "spike" around the L3 cache size for the no-partition case. Counter-intuitively, the execution time of many benchmarks decreases when the working-set size of the interference exceeds the size of the L3. We observe similar spikes on all tested boards. On the Nvidia Orin, the spike is observed even when interference and benchmarks execute on different clusters, *i.e.,* when only the system L4 cache is shared. However, such spikes are not observed in either set or way partitioned scenarios. To further investigate this effect, we analyzed the Cortex-A55 PMCs `L1D_CACHE_REFILL_INNER` and `L1D_CACHE_REFILL_OUTER` (Sec. IV-C). Fig. 7 shows the two PMCs for *mser/vga* benchmark. `L1D_CACHE_REFILL_INNER` dips at the L3 cache size and rises again, indicating that more L3 is allocated to *mser* as the interference size increases. In contrast, the PMC values are nearly constant for a partitioned cache, indicating a constant usage of the L3 cache. The spikes correlate, therefore, with the interaction between interference and benchmarks. Below the L3 cache size, the fast loops of the interference workload are almost always cache-hot in L1, L2, and L3 caches. Cache lines used by the benchmark are quickly evicted by the interference. As noted in Obs. 5, memory accesses increase sharply after the L3 cache size, leading to overall higher latencies and slower execution of the requests that must be served from memory. Interference-related memory accesses are also served slower, leading to less frequent evictions for the benchmarks. In turn, this yields faster execution time for the benchmarks and the decrease of the slowdown ratio. Instead, benchmarks and interference don't share common cache lines in the partitioned cache scenarios, leading to a constant execution time as the memory controller bandwidth saturates to the highest value.

***Obs. 7:* Way partitioning is less effective on Orin**

Fig. 8 shows the slowdown ratio of *disparity/cif* with *read* interference for RK3568, RK3588, and Orin. On Orin, we run the vision-benchmark on one core and the interference-bench



(a) RK3568

(b) RK3588

(c) Orin

Fig. 8: Slowdown of *disparity/cif* under read interference on RK3568, RK3588, and Orin with way partitioning.

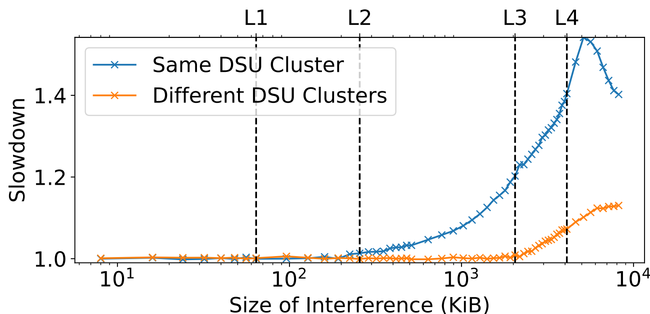


Fig. 9: Slowdown of *mser/vga* on Orin under *modify* interference running on same and different DSU clusters.

on another core in the *same* cluster. As visible in Fig. 8c, the benefits of way partitioning on Orin are lower than on RK3568 (Fig. 8a) and RK3588 (Fig. 8b). In particular, on Orin, the **1/3** and **2/2** cases produce a higher slowdown than without partitioning. Such slowdown increases considerably as we cross the L3 cache size. The lower effectiveness of way partitioning can be attributed to the presence of the non-partitioned system cache (L4) in the Orin. Here inter-core interference is unregulated.

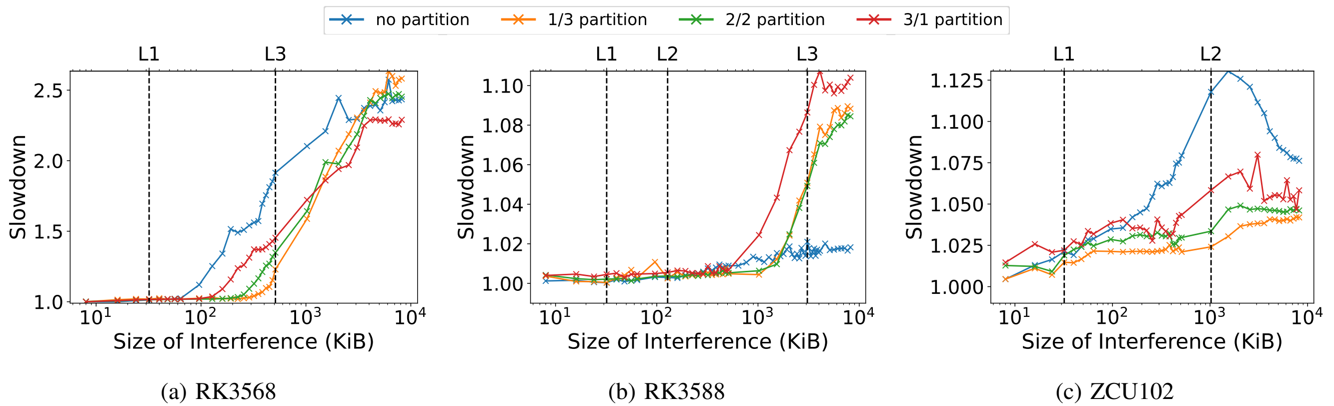
Obs. 8: **Interference within DSU clusters is significantly higher than across clusters**
On the NVIDIA Orin, we measure the slowdown when the interference workload and the benchmark application run on different DSU clusters. The Orin is the only tested platform that features multiple DSU clusters and an additional system-level L4 cache. Fig. 9 presents the slowdown of *mser/vga* under *modify* interference. As expected, in the different cluster configuration, there is no measurable slowdown when the interference workload fits entirely within the L3 cache. However, even when the interference workload uses the L4 cache, the slowdown of this configuration is minimal compared to the same cluster one. We attribute this behavior to the different pressure on the L4 cache in the two configurations. Due to the exclusive availability of the L3 cache in the different-cluster configuration, the pressure on the L4 cache is lower than in the same-cluster one.

Obs. 9: **Partitioning does not protect against streaming *write* interference**
The A55 cores support a *write-streaming* mode, where continuous store operations bypass the cache. This minimizes cache pollution by not allocating data that is unlikely to be accessed soon. *Write-streaming* is controlled from the `CPUECTRL` register. By default, the A55 cores apply *write-streaming* mode to the 4th, 128th, and 1024th consecutive streaming write access on the L1, L2, and L3 caches, respectively [50]. With a cache line size of 64 B, after a continuous write of 64 KB, data does not allocate to the L3 cache. On RK3568 and RK3588, in *write-streaming* mode, cache partitioning techniques do not improve the slowdown ratio compared to a case without partitioning.

Fig. 10 illustrates the slowdown (w.r.t. no-interference) of *disparity/cif* on RK3568, RK3588, and ZCU102 when the interference workload is *write*. As can be observed in Fig. 10b for the RK3588 platform (and in Table I on the RK3568), under *write* interference, partitioning techniques can produce higher slowdowns. In contrast, as shown in Fig. 10c, on the ZCU102—which does not have *write-streaming*—we observe that partitioning improves the slowdown ratio. In fact, on the RK3568 and RK3588 when caches are not partitioned, the maximum interference in *write-streaming* mode is limited to 64 KB. However, in cases **1/3**, **2/2** and **3/1**, we force a reduction of the cache available to the benchmark by a size much larger than a 64 KB reduction. The remaining source of interference comes from contention in memory accesses, which saturates to the same value in both partitioned and non-partitioned (see Fig. 6).

Cache partitioning is therefore ineffective in *write-streaming* mode since caches and memory controller interference are anyway less polluted. To evaluate further, we repeated the experiment by disabling the *write-streaming* mode on the RK3568 platform. Fig. 11a and Fig. 11b show the effect of *write* on *mser/vga* benchmark, when *write-streaming* mode, is enabled and disabled respectively. From Fig. 11b, we observe that disabling *write-streaming* mode makes cache partitioning effective in reducing slowdowns.

Obs. 10: **Platform Specific Observations**
The slowdown ratio increases around the L1 cache size in RK3568 and ZCU102 platforms irrespective of cache parti-

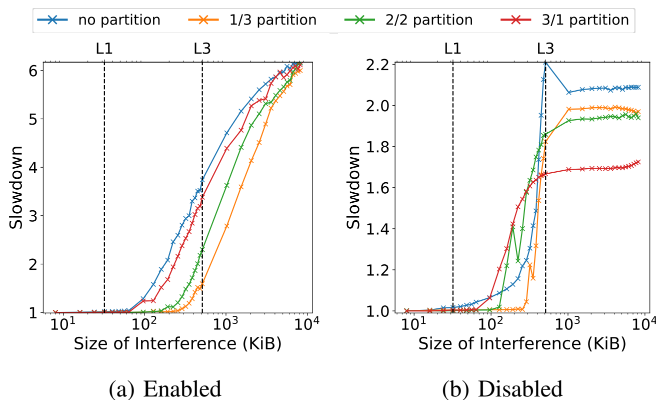


(a) RK3568

(b) RK3588

(c) ZCU102

Fig. 10: Slowdown of *disparity/cif* under *write* interference on RK3568, RK3588, and ZCU102 with set partitioning.



(a) Enabled

(b) Disabled

Fig. 11: Slowdown of *mser/vga* under write interference with *write-streaming* mode a) Enabled and b) Disabled on RK3568

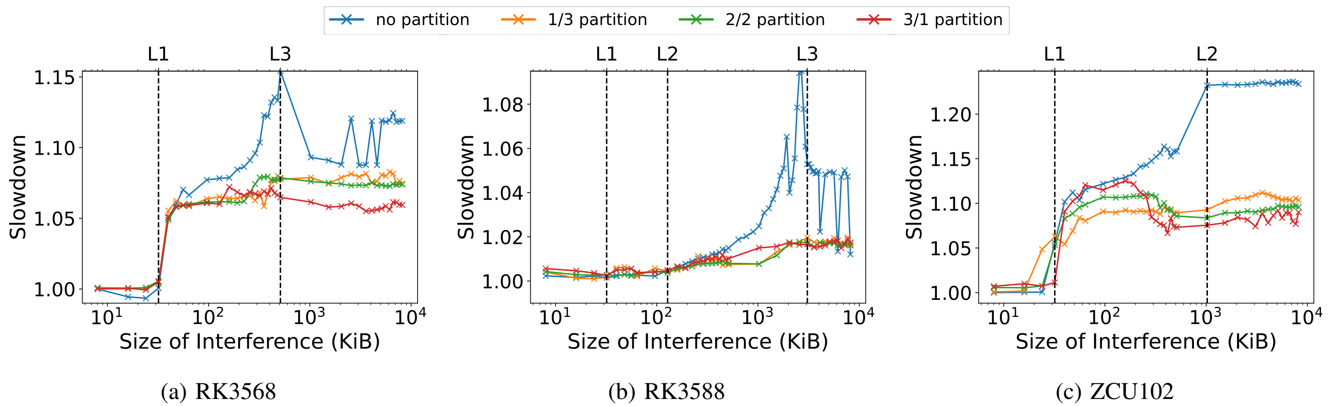
tioning. When the size of the interference benchmark exceeds the size of the L1 cache, benchmark's execution times on RK3568 and ZCU102 increase. This is clearly visible in Fig. 12 where the execution time jumps by 6% and 10% on RK3568 and ZCU102, respectively. This effect is also present in the *tracking* and *mser* datasets for *read* and *modify* cases, but not for *prefetch* in RK3568. This slowdown is not observed on the RK3588 or Orin platforms, likely due to the presence of a private L2 cache mitigating the effect. However, there is no conclusive evidence from the performance counters on any platform to fully explain this behavior.

Slowdown from write interference is much higher on RK3568. From a cache point of view, write operations are expensive compared to read since they necessitate invalidating the cacheline, maintaining coherency across memory hierarchies, and writing back to the slower memory. *write* produces much worse slowdowns compared to *read/modify/prefetch* on RK3568. Instead, on RK3588, performance is comparable (Table I and Fig. 10a 10b). We suspect the higher slowdowns on RK3568 to be a side effect of the *write-streaming* mode. When *write-streaming* mode is disabled, the slowdown decreases on the RK3568 platform. The maximum slowdown reduced from 6.2x to 2.2x in a non-partitioned cache for the *mser/vga* dataset, and for a **3/1** way-partitioned cache with the same dataset, the slowdown reduced from 6.1x to 1.72x (Fig. 11).

In write-streaming *mode on RK3568 and RK3588 platforms, performance counters indicate allocation into the L3 cache.* Using the `L3D_WS_MODE` A55 performance counter (Sec. IV-C), one can monitor the transition into *write-streaming* mode in the L3 cache. If we make continuous write accesses to a size greater than 64 KB, we observe a high value in this counter, confirming that the core enters a write-streaming mode. However, `L3D_CACHE_ALLOCATE` performance counter in the DSU shows comparable values when *write-streaming* mode is enabled or disabled (by modifying the `CPUECTRL` register). This counters the expectation that `L3D_CACHE_ALLOCATE` does not count in write-streaming mode as the data accessed are not being allocated to the cache. We are not sure if this is an error in the behavior of *write-streaming* mode or counting implementation of `L3D_CACHE_ALLOCATE` counter.

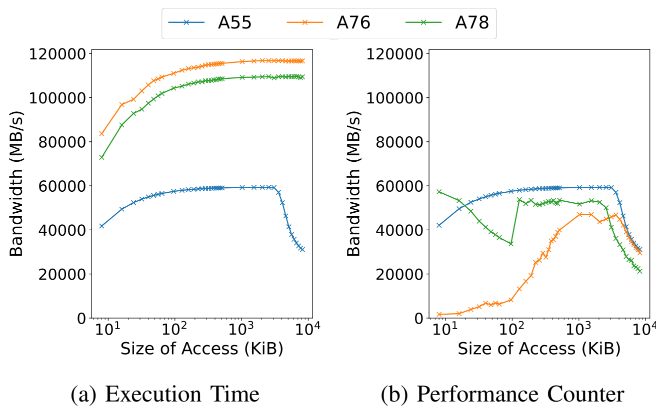
The effect of prefetch is stronger on the A55 cores than on the A76 or A78 cores. We compare the bandwidth of *prefetch* accesses obtained from the DSU `SCU_PFTCH_CPU_ACCESS` counter and from the execution time of the program. As seen in Fig. 13, on the A55 cores, the execution time is higher, but the bandwidth obtained from it matches the bandwidth obtained from performance counters. On the A76 and the A78 cores, the execution time is much lower, but the bandwidth obtained from the execution time is much higher than that obtained from the performance counter. This might imply that the A76 and A78 cores take the prefetch instructions as more of a hint, while the A55 cores execute it and load the contents to the cache as specified in the software prefetch command.

Summary: Our findings indicate that both way and set partitioning techniques effectively reduce interference between cores (Obs. 2), but neither consistently outperforms the other across the tested benchmarks (Obs. 3). Way-partitioning is simpler to implement and remains unaffected by cache scrambling mechanisms that could hinder set-partitioning (Sec. IV-F); however, it can increase conflict misses under specific access patterns (Obs. 1). Regardless of the method used, cache partitioning increases pressure on the memory controller, even when the workload is smaller than the L3 cache, potentially degrading overall performance (Obs. 5). Fur-



(a) RK3568 (b) RK3588 (c) ZCU102

Fig. 12: Slowdown of *disparity/cif* under *modify* interference on RK3568, RK3588, and ZCU102 with set partitioning.



(a) Execution Time (b) Performance Counter

Fig. 13: Bandwidth of executing *prefetch* calculated from a) Execution time of program b) Performance Counters

thermore, architectural optimizations, such as write-streaming, can diminish the effectiveness of cache partitioning, with their impact varying across different SoCs due to implementation differences (Obs. 9).

The tests also show that increasing the interference workload size without cache partitioning leads to execution time spikes (Obs. 6). This behavior originates from the memory controller becoming overloaded, reducing the capability of the interference-bench to evict cache lines of the tested benchmarks. The effect disappears with cache partitioning since benchmarks and interference do not share cache lines.

Our results on the Orin—the only platform tested with multiple DSU clusters—highlight that, despite way-partitioning at the L3 cache, interference is significant due to the shared, unpartitioned L4 cache across DSU clusters (Obs. 7). However, interference between cores in different DSU clusters is much lower than within the same cluster, emphasizing the benefits of cluster-level isolation (Obs. 8).

VI. Conclusion and Future Directions

This paper presented a comprehensive evaluation of Arm's DSU technology for real-time cache partitioning across multiple COTS platforms. Through detailed testing, we examined the capabilities and limitations of Arm DSU's way partitioning feature, comparing it with established set partitioning techniques such as *page coloring*. Our findings show that way partitioning provides effective isolation with lower setup overhead, making it a practical option for reducing interference in real-time applications. Conversely, set partitioning offers finer control by allowing selective cache partitioning, not only at the last-level cache (LLC) but also at higher hierarchy levels (*e.g.,* L1, L2). This flexibility is particularly beneficial on platforms like NVIDIA Orin, where managing interference at both the L3 and L4 cache levels can improve performance and predictability.

Overall, this study identifies the trade-offs practitioners should consider when selecting a cache partitioning strategy for real-time systems. While way-partitioning offers ease of implementation and resilience against hardware constraints, its effectiveness depends on workload characteristics and system-level optimizations. Understanding platform-specific architectural behaviors—such as memory controller saturation, prefetching strategies, and write-streaming mechanisms, is needed for achieving predictable performance.

This study focuses exclusively on inter-core interference. We defer to future work the effective integration of cache partitioning at scheduling level to manage interference among different tasks. Here, both way and set partitioning could be leveraged for task-level isolation, but their efficiency in this context requires further investigation. While way-partitioning can be dynamically adjusted by modifying the `CLUSTERPARTCR` register, the potential overhead of frequent changes has not been studied. Future work should assess whether this overhead is lower than the one incurred in set-partitioning.

Acknowledgments

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

References

[1] Arm, "Arm big.LITTLE Technology," https://www.arm.com/technologies/big-little Accessed: 2024-11-09.

[2] Anandtech, "Intel Hybrid Performance," https://www.anandtech.com/print/17047/the-intel-12th-gen-core-i912900k-review- hybrid-performance-brings-hybrid-complexity Accessed: 2024-11-09.

[3] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multi-core architectures,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, p. 45–54.
[4] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013, pp. 55–64.
[5] 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)*, 2014, p. 155–166.
[6] 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)*, 2016, pp. 1–12.
[7] Intel, “Resource Director Technology,” https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html Accessed: 2024-11-09.
[8] Arm, “Quality of Service in ARM Systems: An Overview,” https://community.arm.com/arm-community-blogs/b/soc-design-and-simulation-blog/posts/quality-of-service-in-arm-systems-an-overview Accessed: 2024-11-09.
[9] ——, “Arm Memory System Resource Partitioning and Monitoring (MPAM) System Component Specification,” https://developer.arm.com/documentation/ihi0099/ Accessed: 2024-11-09.
[10] 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)*, 2019, p. 345–356.
[11] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, “E-WarP: A System-wide Framework for Memory Bandwidth Profiling and Management,” in *2020 IEEE Real-Time Systems Symposium (RTSS)*, 2020.
[12] G. Schwaericke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, “A Real-Time virtio-based Framework for Predictable Inter-VM Communication,” in *2021 IEEE International Real-Time Systems Symposium (RTSS)*, 2021.
[13] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger, “A Closer Look at Intel Resource Director Technology (RDT),” in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 127–139. [Online]. Available: https://doi.org/10.1145/3534879.3534882
[14] M. Zini, D. Casini, and A. Biondi, “Analyzing Arm’s MPAM from the perspective of time predictability,” *IEEE Trans. Computers*, vol. 72, no. 1, pp. 168–182, 2023. [Online]. Available: https://doi.org/10.1109/TC.2022.3202720
[15] Arm, “Arm DynamIQ Shared Unit Technical Reference Manual,” https://developer.arm.com/documentation/100453/ Accessed: 2024-11-09.
[16] Rockchip, “Rockchip RK3568,” https://www.rock-chips.com/a/en/products/RK35_Series/2021/0113/1276.html Accessed: 2024-11-09.
[17] ——, “Rockchip RK3588,” https://www.rock-chips.com/a/en/products/RK35_Series/2022/0926/1660.html Accessed: 2024-11-09.
[18] NVIDIA, “NVIDIA Jetson AGX Orin,” https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/ Accessed: 2024-11-09.
[19] AMD, “Zynq UltraScale+ Device Technical Reference Manual,” https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual.
[20] 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)*. IEEE, 2009, pp. 55–64.
[21] S. AG, “Jailhouse hypervisor,” https://github.com/siemens Accessed: 2024-11-09.
[22] 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)*, 2019, p. 1–14.
[23] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, “Wcet(m) estimation in multi-core systems using single core equivalence,” in *27th Euromicro Conference on Real-Time Systems, ECRTS 2015, Lund, Sweden, July 8-10, 2015*. IEEE Computer Society, 2015, pp. 174–183. [Online]. Available: https://doi.org/10.1109/ECRTS.2015.23
[24] L. Sha, M. Caccamo, R. Mancuso, J. Kim, M. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. R. Perlman, G. Arundale, and R. M. Bradford, “Real-time computing on multicore processors,” *Computer*, vol. 49, no. 9, pp. 69–77, 2016. [Online]. Available: https://doi.org/10.1109/MC.2016.271
[25] T. Lugo, S. Lozano, J. Fernández, and J. Carretero, “A survey of techniques for reducing interference in real-time applications on multicore platforms,” *IEEE Access*, vol. 10, pp. 21 853–21 882, 2022. [Online]. Available: https://doi.org/10.1109/ACCESS.2022.3151891
[26] 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, 2015. [Online]. Available: https://doi.org/10.1145/2830555
[27] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9-13 December 2006, Orlando, Florida, USA*. IEEE Computer Society, 2006, pp. 423–432. [Online]. Available: https://doi.org/10.1109/MICRO.2006.49
[28] Y. Xie and G. H. Loh, “PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches,” in *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, S. W. Keckler and L. A. Barroso, Eds. ACM, 2009, pp. 174–183. [Online]. Available: https://doi.org/10.1145/1555754.1555778
[29] P. D. Halwe, S. Das, and H. K. Kapoor, “Towards a better cache utilization using controlled cache partitioning,” in *IEEE 11th International Conference on Dependable, Autonomic and Secure Computing, DASC 2013, Chengdu, China, December 21-22, 2013*. IEEE Computer Society, 2013, pp. 179–186. [Online]. Available: https://doi.org/10.1109/DASC.2013.59
[30] P. Das, N. M. Barbhuiya, and B. Ranjan Roy, “A survey on way-based cache partitioning,” in *2023 IEEE Silchar Subsection Conference (SILCON)*, 2023, pp. 1–7.
[31] S. Srikantaiah, M. T. Kandemir, and M. J. Irwin, “Adaptive set pinning: managing shared caches in chip multiprocessors,” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, S. J. Eggers and J. R. Larus, Eds. ACM, 2008, pp. 135–144. [Online]. Available: https://doi.org/10.1145/1346281.1346299
[32] V. Suhendra and T. Mitra, “Exploring locking & partitioning for predictable shared caches on multi-cores,” in *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, L. Fix, Ed. ACM, 2008, pp. 300–303. [Online]. Available: https://doi.org/10.1145/1391469.1391545
[33] R. R. Iyer, “Cqos: a framework for enabling qos in shared caches of CMP platforms,” in *Proceedings of the 18th Annual International Conference on Supercomputing, ICS 2004, Saint Malo, France, June 26 - July 01, 2004*, P. Feautrier, J. R. Goodman, and A. Seznec, Eds. ACM, 2004, pp. 257–266. [Online]. Available: https://doi.org/10.1145/1006209.1006246
[34] A. Wolfe, “Software-based cache partitioning for real-time applications,” *J. Comput. Softw. Eng.*, vol. 2, no. 3, p. 315–327, Mar. 1994.
[35] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, “Outstanding paper award: Making shared caches more predictable on multicore platforms,” in *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*. IEEE Computer Society, 2013, pp. 157–167. [Online]. Available: https://doi.org/10.1109/ECRTS.2013.26
[36] N. Kim, B. C. Ward, M. Chisholm, C. 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), Vienna, Austria, April 11-14, 2016*. IEEE Computer Society, 2016, pp. 149–160. [Online]. Available: https://doi.org/10.1109/RTAS.2016.7461323

[37] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: a dynamic cache partitioning system using page coloring," in *International Conference on Parallel Architectures and Compilation, PACT '14, Edmonton, AB, Canada, August 24-27, 2014*, J. N. Amaral and J. Torrellas, Eds. ACM, 2014, pp. 381–392. [Online]. Available: https://doi.org/10.1145/2628071.2628104

[38] Xilinx, "Xilinx Xen Support with Cache-Coloring," https://github.com/Xilinx/xen/releases/tag/xilinx-v2020.2 Accessed: 2024-11-09.

[39] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Design Autom. Electr. Syst.*, vol. 21, no. 3, pp. 49:1–49:24, 2016. [Online]. Available: https://doi.org/10.1145/2858792

[40] Arm, "Corelink Level 2 Cache Controller L2C-310," https://developer.arm.com/documentation/ddi0246/h/functional-overview/cache-operation/cache-lockdown Accessed: 2024-11-09.

[41] M. Hassan, "Reduced latency DRAM for multi-core safety-critical real-time systems," *Real-Time Systems*, pp. 1–36, 2019.

[42] F. Farshchi, Q. Huang, and H. Yun, "BRU: Bandwidth Regulation Unit for Real-Time Multicore Processors," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 364–375.

[43] P. K. Valsan and H. Yun, "MEDUSA: A Predictable and High-Performance DRAM Controller for Multicore Based Embedded Systems," in *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, 2015, pp. 86–93.

[44] M. Paolieri, E. Quiñones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embed. Syst. Lett.*, vol. 1, no. 4, pp. 86–90, 2009. [Online]. Available: https://doi.org/10.1109/LES.2010.2041634

[45] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM controller: bank privatization for predictability and temporal isolation," in *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*, R. P. Dick and J. Madsen, Eds. ACM, 2011, pp. 99–108. [Online]. Available: https://doi.org/10.1145/2039370.2039388

[46] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "DRAMbulism: Balancing Performance and Predictability through Dynamic Pipelining," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 82–94.

[47] A. Zuepke, A. Bastoni, W. Chen, M. Caccamo, and R. Mancuso, "Mempol: Policing core memory bandwidth from outside of the cores," in *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*. IEEE, 2023, pp. 235–248. [Online]. Available: https://doi.org/10.1109/RTAS58335.2023.00026

[48] AMD, "AMD64 Technology Platform Quality of Service Extensions," https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/other/56375_1_03_PUB.pdf Accessed: 2024-11-14.

[49] Arm, "Arm Cortex-A53 MPCore Processor Technical Reference Manual," https://developer.arm.com/docs/ddi0500/ Accessed: 2024-11-09.

[50] ——, "Arm Cortex-A55 Core Technical Reference Manual," https://developer.arm.com/docs/100442/ Accessed: 2024-11-09.

[51] ——, "Arm Cortex-A76 Core Technical Reference Manual," https://developer.arm.com/docs/100798/ Accessed: 2024-11-09.

[52] ——, "Arm Cortex-A76AE Core Technical Reference Manual," https://developer.arm.com/docs/101392/ Accessed: 2024-11-09.

[53] ——, "Arm Cortex-A78 Core Technical Reference Manual," https://developer.arm.com/docs/101430/ Accessed: 2024-11-09.

[54] ——, "Arm Cortex-A78AE Core Technical Reference Manual," https://developer.arm.com/docs/101779/ Accessed: 2024-11-09.

[55] ——, "Arm DynamIQ Shared Unit-AE Technical Reference Manual," https://developer.arm.com/documentation/101322/ Accessed: 2024-11-09.

[56] M. Systems, "Memory-aware Jailhouse hypervisor," https://github.com/Minervasys/jailhouse Accessed: 2024-11-09.

[57] Xilinx, "Xilinx 6.1_LTS rebased Linux," https://github.com/Xilinx/linux-xlnx/tree/xlnx_rebase_v6.1_LTS.

[58] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2020, Sydney, Australia, April 21-24, 2020*. IEEE, 2020, pp. 296–309. [Online]. Available: https://doi.org/10.1109/RTAS48715.2020.00006

[59] D. Costa, L. Cuomo, D. Oliveira, I. M. Savino, B. Morelli, J. Martins, F. Tronci, A. Biasci, and S. Pinto, "IRQ coloring: Mitigating interrupt-generated interference on ARM multicore platforms," in *Fourth Workshop on Next Generation Real-Time Embedded Systems, NG-RES 2023, January 18, 2023, Toulouse, France*, ser. OASIcs, F. Terraneo and D. Cattaneo, Eds., vol. 108. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 2:1–2:13. [Online]. Available: https://doi.org/10.4230/OASIcs.NG-RES.2023.2

[60] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, "Rt-bench: An extensible benchmark framework for the analysis and management of real-time applications," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 184–195. [Online]. Available: https://doi.org/10.1145/3534879.3534888

[61] A. Pradhan, D. Ottaviano, Y. Jiang, H. Huang, A. Zuepke, A. Bastoni, and M. Caccamo, "Arm DynamIQ Shared Unit and Real-Time: An Empirical Evaluation - Extended Version," *arXiv preprint arXiv:2503.17038*, 2025. [Online]. Available: https://arxiv.org/pdf/2503.17038

[62] S. A. Panchamukhi and F. Mueller, "Providing task isolation via tlb coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 3–13.