

MEMSCOPE: Open-Source Kernel-Level Framework for Heterogeneous Memory Characterization

Golsana Ghaemi
Boston University

Boston, Massachusetts, USA
golsana@bu.edu

Gabriel Franco
Boston University

Boston, Massachusetts, USA
gvfranco@bu.edu

Kazem Taram
Purdue University

West Lafayette, Indiana, USA
kazem@purdue.edu

Renato Mancuso
Boston University

Boston, Massachusetts, USA
rmancuso@bu.edu

Abstract—This paper presents an open-source kernel-level heterogeneous memory characterization framework (MEMSCOPE) for embedded systems. MEMSCOPE enables precise characterization of the temporal behavior of available memory modules under configurable contention stress scenarios. MEMSCOPE leverages kernel-level control over physical memory allocation, cache maintenance, CPU state, interrupts, and I/O device activity to accurately benchmark heterogeneous memory subsystems. This gives us the privilege to directly map pieces of contiguous physical memory and instantiate allocators, allowing us to finely control cores to create and eliminate interference. Additionally, we can minimize noise and interruptions, guaranteeing more consistent and precise results compared to equivalent user-space solutions. Running our Framework on a Xilinx Zynq UltraScale+ ZCU102 CPU-FPGA platform demonstrates its capability to precisely benchmark bandwidth and latency across various memory types, including PL-side DRAM and BRAM, in a multi-core system.

Index Terms—heterogeneous memory, benchmarking, resource management, multi-core real-time systems.

I. INTRODUCTION

The ever-increasing demand for high-performance systems, combined with the steady rise in data-intensive processing workloads, has been a defining force for the modern landscape of hardware platforms. The push for higher performance has impacted general-purpose systems and embedded/real-time systems. System *heterogeneity* has been pivotal in the last decade of embedded systems evolution and the subject of a plethora of studies [1]. Modern high-performance systems-on-chip (SoCs) are characterized by high **compute heterogeneity**. Indeed, they consist of a wide range of cross-vendor computing blocks ranging from general-purpose processors (CPUs) to special-purpose accelerators and even FPGAs. Established OS-level methodologies have emerged to benchmark and support application development in heterogeneous systems. Notable examples include the Linux Remote Processor Framework [2] and the OpenMP Framework [3].

The heterogeneity in modern platforms is not limited to computing resources. **Memory heterogeneity** has co-evolved with compute heterogeneity. Different memory technologies coexist, each with specific characteristics in terms of size, cost, and temporal behavior. Not only do the baseline performance (e.g., single-threaded accesses) of these memories range widely, but so do their temporal behavior under stress (e.g., multi-threaded accesses). Notable examples of memory technologies with widely ranging characteristics include Double

Data Rate (DDR), Reduced-Latency DRAM (RL-DRAM) [4], High-Bandwidth Memory (HBM) [5], Non-Volatile Random Access Memory (NVRAM) [6], and on-chip Static Random Access Memory (SRAM) [7], [8].

Challenges. We focus on memory heterogeneity. While heterogeneous memory subsystems present vast opportunities to optimize memory allocation for real-time and embedded applications, their practical use presents several challenges. Said challenges can be grouped into *Characterization Challenges* and *Usage Challenges*. Characterization challenges hinder the construction and deployment of precise, controlled, and interference-free experiments to understand the temporal behavior of memory modules when relying on conventional user-space toolkits. Usage challenges impede the efficient allocation of heterogeneous memory to user-space applications.

MEMSCOPE as the Proposed Solution. In this paper, we address characterization challenges. To do so, we design, implement, and evaluate a novel open-source in-kernel heterogeneous memory characterization toolkit called MEMSCOPE. MEMSCOPE is designed as a Linux kernel module, requiring no kernel source modifications, to boost broad adoption. It is designed to (1) automatically recognize heterogeneous memory modules described via the kernel device tree; (2) internally instantiate per-memory allocators under the direct control of system evaluators; (3) provide an extensible library of micro-benchmarking activities; (4) allow intuitive experiment definition and results retrieval from user-space; and (5) minimize experimental noise with direct control over CPU and interrupt state during an active experiment.

Contribution. This paper makes the following contributions. (1) We propose the first kernel-level heterogeneous memory characterization framework, namely MEMSCOPE; (2) We provide a full open-source implementation of MEMSCOPE; (3) We evaluate the capabilities of MEMSCOPE on a modern embedded platform featuring a high degree of memory heterogeneity; (4) We demonstrate that MEMSCOPE allows accurate characterization with valuable insights to drive memory allocation in user-space applications.

II. MOTIVATION AND GOAL

Attention to memory management in heterogeneous systems has received substantial interest from the general-purpose and high-performance systems computing community, as we

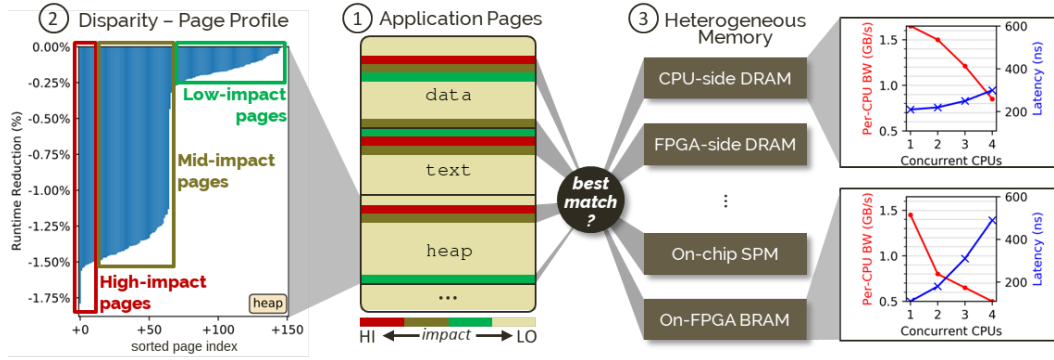


Fig. 1. The problem of heterogeneous memory management consists in performing memory allocation given proper characterization of the temporal behavior of memory modules due to technological heterogeneity (right-hand side), and the expected impact of a given memory page on the temporal behavior of applications due to usage heterogeneity (left-hand side).

review in Section V. Nonetheless, no *de facto* turnkey solution exists to perform heterogeneous memory characterization efficiently. MEMSCOPE aims to fill this gap, primarily targeting Linux-based high-performance real-time embedded systems.

A. Sources of Memory Heterogeneity

Heterogeneous memory subsystems amplify the complexity of proper management for time-sensitive applications due to the interplay of two effects, namely *technological heterogeneity* as shown in Figure 1 inset (3) and *usage heterogeneity*, as depicted in Figure 1 inset (2).

Technological Heterogeneity. As briefly mentioned in Section I, memory modules differ in size, cost, and inherent temporal characteristics, such as read/write latency and bandwidth. Several hardware-level characteristics contribute to the exhibited temporal characteristics, such as (1) the type of memory cells they comprise (SDRAM, SRAM, or NVRAM) which impacts their performance, power, and persistence characteristics; (2) their architectural organization—e.g., SDRAM cells can be flatly arranged to in traditional DRAM systems, or 3D-stacked in High-bandwidth Memory (HBM) modules; SRAM cells can be used to define architectural caches, scratchpads, or in FPGAs as Block RAM (BRAM) and ultraBRAM modules.

Moreover, different memory types exhibit varying performance characteristics under contention, owing to their intrinsic memory-level parallelism (MLP). As such, bandwidth and latency can be impacted by interference from concurrent tasks or competing memory requests from multiple cores, leading to nonlinear performance degradation. As depicted on the right-hand side of Figure 1, memory modules in a heterogeneous memory subsystem are characterized by **performance curves** parametrized by the type of accesses and the degree of contention. For instance, traditional CPU-side DRAM might exhibit worse single-threaded latencies than an FPGA-side scratchpad (BRAM) but sustain better multi-threaded bandwidth as concurrent accesses increase.

Usage Heterogeneity. Memory resources are often the performance bottleneck in data-heavy workloads. Depending on the application, low-latency access to some memory pages might largely impact the execution time. Conversely, placing other pages in slow memory might have a negligible impact. Fortunately, the need to profile the demand of applications for memory resources is well understood [9]–[11]. Borrowing

and annotating a figure from [11], the left-hand side of Figure 1 depicts the per-page runtime reduction percentage when individual heap pages are allocated in cache.

B. Key Challenges

Inspired by the famous quote “*You can’t manage what you don’t measure*,” often attributed to Peter Drucker, we aim to systematically analyze the temporal characteristics of heterogeneous memory subsystems in embedded systems, gather deeper insights into performance variations and optimize memory usage. To this end, we propose an extensible and easy-to-use kernel-based benchmarking infrastructure addressing the key challenges (C1–C5) reviewed below.

C1: Imprecise Physical Memory Allocation. In user-space, memory allocation is mediated by the virtual memory layer. Thus, limited control can be exerted over physical memory allocation. This shortcoming poses a fundamental challenge when characterizing heterogeneous memory.

C2: Imprecise Compute Engine Activity. To evaluate memory performance under isolated conditions, one must control the execution context of the benchmarking activities. In user-space, it is challenging to prevent system daemons, kernel threads, and background processes from interfering.

C3: Imprecise Interrupt Activity. In user-space, applications cannot disable or redirect interrupts, nor can they prevent the kernel or other subsystems from servicing them on the core of interest. This leads to two major sources of noise: (1) interrupts can preempt benchmarking tasks, and (2) servicing interrupts may generate additional memory traffic.

C4: Restricted Cache Maintenance. Caches often act as an opaque layer that masks the true behavior of the underlying memory. Thus, controlling cache states is key to accurately assessing memory performance. User-space applications, however, are restricted in their access to cache maintenance instructions or cache-control interfaces.

C5: Restricted Access to Performance Counters. Hardware performance counters offer fine-grained visibility into metrics that are invaluable when dissecting the behavior of complex memory systems. Unfortunately, user-space access to performance counters is often limited or highly abstracted.

C. The MEMSCOPE Approach

To attain full control over allocation strategies, access patterns, cache invalidation, access to performance monitors, and

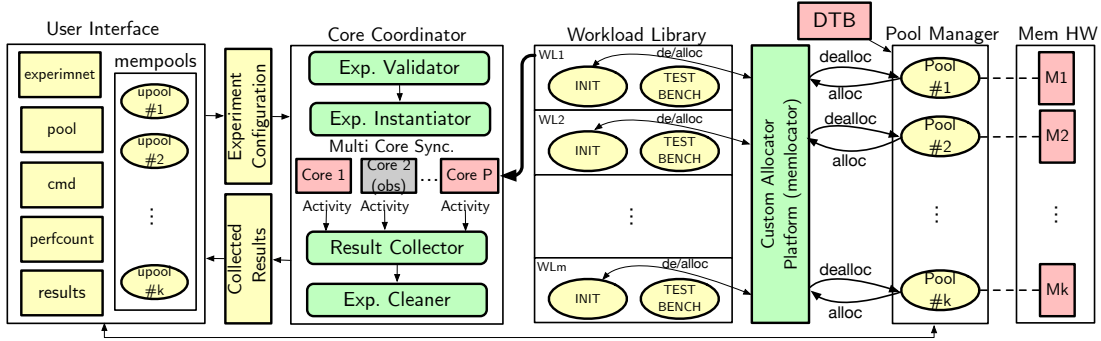


Fig. 2. High-level structure of MEMSCOPE highlighting its main components and their interplay.

CPU states, we implement our benchmarking infrastructure at the kernel level. A similar motivation fueled the seminal work on NanoBench [12], the only kernel-level toolkit for single-core CPU benchmarking. MEMSCOPE is the first kernel-level toolkit for the characterization of heterogeneous memory subsystems in multicore systems.

III. MEMSCOPE DESIGN

In this section, we describe the primary design elements of MEMSCOPE. The overall system design, depicted in Figure 2, comprises four main components. First, we cover the structure of each benchmarking experiment in MEMSCOPE—see Section III-A. Next, we discuss the various sub-modules depicted in Figure 2 that are crucial for the following functionalities: (1) memory target selection via a *Memory Pool Manager* (Section III-B); (2) access pattern selection via the *Workload Library* (Section III-C); (3) multi-CPU orchestration via the *Core Coordinator* (Section III-D); (4) user interaction for experiment control and result retrieval (Section III-E). The full code and implementation details are available in our repository¹.

On top of what was mentioned above, due to the widely varying temporal characteristics of different memories, gaining insight into these variations allows us to understand how application runtimes are impacted by allocation decisions, especially crucial for safety-critical real-time embedded systems. Certain pages will experience more or less interference depending on allocation strategies, affecting overall performance. By analyzing these behaviors, we can make informed decisions about memory allocation to mitigate contention and optimize execution. Our work opens the door for future integration into memory allocation, utilizing heterogeneous memories. In Section III, we will outline the blueprint of our approach, explain the design challenges, and how we tackle these challenges.

A. Experiment Structure in MEMSCOPE

The goal of a MEMSCOPE experiment is to evaluate the temporal characteristics of a target memory module under a varying degree of contention generated by the other online CPUs. As such, each experiment in MEMSCOPE consists of a sequence of *scenarios*. Each scenario is comprised of a set of *monitored activities* across all online CPUs. All experiments follow a common structure:

- 1) Memory targets and access pattern parameters for the core under observation and stressor cores are runtime configurable.
- 2) The temporal behavior of the observed core is measured following a sequence of increasingly high-stress scenarios.
- 3) Scenario-specific workloads are assigned to both the core under observation and all the interfering cores. The workload assigned to the core under analysis can differ from the one executed by a stressor core.
- 4) Micro-architectural events are collected for all CPUs.
- 5) Results include the total bytes read/written from/to the target memory, the execution time for the core under observation, and the sampled architectural events across all cores.
- 6) At the end of each scenario, and also upon the completion of the entire experiment, MEMSCOPE performs per-core data structure management and deallocates all allocated buffers to ensure a clean state for subsequent experiments.

Scenarios, ranging from the *best* to *worst* case, are executed in an automated sequence. In the best scenario, the core under observation runs the selected workload while all other cores remain *memory-idle* by executing a CPU-intensive, non-memory workload. Once this scenario completes and the results are collected, the second scenario begins: one additional core starts executing the stress workload while the rest remain *memory-idle*. In the following scenario, MEMSCOPE increases the number of stress cores by one. This process continues until the worst-stress scenario: all available cores are actively stressing the selected target memory.

B. Memory Pool Manager

A benchmarking infrastructure for heterogeneous memory subsystems requires designing mechanisms to precisely select the target memory pools. To this end, MEMSCOPE leverages the same mechanisms that the OS uses to describe hardware resources, i.e., *device trees*, to auto-detect an arbitrary number of available memory areas.

MEMSCOPE creates a set of memory pools, each corresponding to a detected memory module. This one-to-one mapping enables memory targets to be selected through pool identifiers (pool ID), as illustrated in Figure 2. The example presented in Figure 2 depicts the memory pool manager and the creation of pools with IDs #1 to # k from available underlying memory modules $M_1, M_2, \dots, M_k$.

The pool manager eliminates the need for manual detection and configuration of memory pools parameters, enhancing flexibility. This design also allows for the seamless integration

¹<https://github.com/rntmancuso/black-box-profiler/tree/MemScope-release>

of additional memory technologies, e.g., Non-Volatile Memory (NVM) and disaggregated remote memory.

The instantiated memory pools are primarily used internally to conduct memory performance experiments. In addition, MEMSCOPE also exports these pools for memory allocation in user space (*upools*).

C. Workload Library

Apart from selecting the memory to benchmark, MEMSCOPE also allows one to select the performance metric to be measured for the chosen memory target. The specific choice depends on the particular features of the memory subsystem one wish to analyze, as well as the stress/memory contention scenarios for which insights are desired. It is important to note that depending on the experiment parameters, MEMSCOPE allows to benchmark not only the target memory module, but also its interplay with CPU caches and bus architecture, as demonstrated in our evaluations—see Section IV.

To this end, the workload library offers a suite of configurable micro-benchmarking workloads, each designed to shed light on a set of specific performance parameters. As such, the included test benches are registered in the library based on the access patterns they implement. This modular approach ensures flexibility and ease of maintenance when expanding or modifying the workload library.

Configurable Buffer Initialization. MEMSCOPE allows the definition of a per-workload buffer initialization routine. This is invoked before activating the corresponding workload to initialize the target memory buffer as needed.

Access Strategies. Our library currently focuses on bandwidth and latency measurements of memories under various access strategies, including (1) normal read, (2) normal write, (3) non-cacheable read, (4) non-cacheable write, (5) non-cacheable write streaming, and (6) read/write with non-temporal load/store instructions. Non-cacheable operations refer to access strategies that bypass the CPU caches, ensuring that read/write operations directly interact with the target memory. These allow measuring the performance of memory modules (e.g., scratchpad) that are smaller than the last-level cache. Finally, non-temporal access patterns are implemented through architectural features that allow specific load/store operations to bypass caches.

Bandwidth Measurement Workloads. The goal of the bandwidth micro-benchmarks included in MEMSCOPE is to estimate the throughput that a target memory module is capable of sustaining at steady state. They generate high-rate memory transactions through sequential accesses at cache-line granularity.

Latency Measurement Workloads. The goal of these workloads is to compute the average round-trip time for a generic memory request. To ensure precise measurements, these workloads must ensure that only one outstanding memory operation at a time is emitted by the core under analysis. By enforcing data dependencies, each access depends on the completion of the previous one, preventing prefetching while ensuring full coverage of the target buffer.

Memory-Idle Workload. In addition to all the mentioned *memory-bound* workloads, MEMSCOPE includes a CPU-bound “busy loop” benchmark to characterize memory-idle conditions by keeping the core inactive in memory.

D. Core Coordinator

When an experiment is launched, the core coordinator is responsible for (1) validating the experiment configuration, (2) deploying all the workloads, (3) managing the synchronization between the cores, and (4) aggregating the final results. Thus, MEMSCOPE’s core coordinator includes two primary components, namely the *Experiment Instantiator* and the *Multi-core Synchronizer*.

Experiment Instantiator. Once the experiment configuration has been received, the instantiator is invoked to sanity-check the configuration and initiate the experiment. The sanity checks concern the configuration parameters in relation to the status of the involved allocation pools. To prevent system crashes and/or experiment failures, MEMSCOPE checks: (1) the validity of requested pool IDs; (2) that the pools have been fully initialized; (3) that any pending deallocations from previous experiments have been completed; (4) that the size of the requested buffers, considering all the active CPUs, does not exceed the size of available per-pool memory; If validation passes, the instantiator spawns the scenario-specific workloads on the online cores. These are referred to as *activities* while they are dispatched on the cores.

MEMSCOPE manages three types of activities: the *Main Activity* on the target core, a selectable *Stress Activity* on stressor cores, and an *Idle Activity* on cores that remain memory-idle using the busy-loop workload. It also handles performance counter sampling and coordinates their configuration per core. Finally, the core coordinator ensures proper synchronization between cores during activity execution which is taken care of by the next submodule.

Multi-Core Synchronization. With p online CPUs, MEMSCOPE measures the temporal behavior of the target memory under selected stress workloads across p scenarios. It ensures that all stressor and idle cores have started their activities before measurements begin on the observed core, preventing overlap that could compromise accuracy and repeatability. Stopping activities also requires synchronization. The core coordinator ensures all cores have fully ceased execution before proceeding, preventing overlap between scenarios and preserving measurement validity.

To ensure accuracy and repeatability, we enforce the following constraints: (1) Measurement on the observed core begins only after all stressor/idle cores have started activity execution; (2) The experimental scenario remains stable throughout the measurement period; (3) Measurement on the observed core stops before any stop command is issued for the other cores; and (4) The next scenario does not begin until all stressor/idle cores have fully completed execution of the previous run. These constraints are strictly enforced by leveraging kernel-level synchronization mechanisms.

E. User-Space Interface

The user-space interface module serves as the primary entry point for interaction with MEMSCOPE to configure, launch experiments, and retrieve results. For this purpose, it exposes a number of entries briefly reviewed below.

Experiment Configuration Entry. Each experiment is configured via a string specifying key parameters: (1) memory mapping type—e.g., normal cacheable; (2) memory access pattern—e.g., sequential for read/write bandwidth measurement; (3) buffer size and (4) target memory pool. Separate parameter sets are provided for the observed core and the stressor cores.

Performance Counter Selection. MEMSCOPE is designed to use all available performance counters, allowing two sets of events to be monitored: one on the observation core and another on the idle or stressor cores.

Pools Status. It provides the list of memory pools detected by MEMSCOPE at load time, including each pool’s ID, size, base physical address, and number of allocatable pages.

Results. This entry provides access to collected results in a user-readable format, including temporal measurements, memory read/write amounts, performance counter values, and the configuration used for the experiment.

Experiment Command. This entry manages experiments, allowing a configured experiment to be started, validated without execution, or erased to release its resources.

F. Portability of MEMSCOPE

MEMSCOPE is designed for broad portability. It can run on any platform that: (1) supports Linux with kernel module insertion, (2) allows reserving memory at boot (e.g., via DTBs on ARM or `mmap` on other architectures), and (3) implements virtual memory via an MMU.

Most of the codebase relies on Linux kernel APIs, making it largely architecture-agnostic. Platform-dependent code is limited to: (1) synthetic testbenches and (2) performance counter interfaces. Testbenches are easily portable across RISC architectures due to ISA similarities, while on CISC systems (e.g., x86) one can leverage known user-space stress techniques such as [13] within MEMSCOPE.

IV. EVALUATION

In this section, we present our evaluation of MEMSCOPE, starting with our methodology and platform setup. The rest of the section presents four classes of experiments: (1) Characterization of DRAM variants and their performance under contended access. (2) Benchmarking of on-chip scratchpad memories to assess their temporal behavior. (3) Analysis of cache microarchitectural behavior and the impact of cache partitioning. (4) Studying the impact of heterogeneous memory management on real-world applications.

A. Experimental Methodology

MEMSCOPE was implemented and tested on Linux kernel v5.4 and evaluated on a Xilinx-ZCU102 development platform featuring a Zynq UltraScale+ XCZU9EG MPSoC [14],

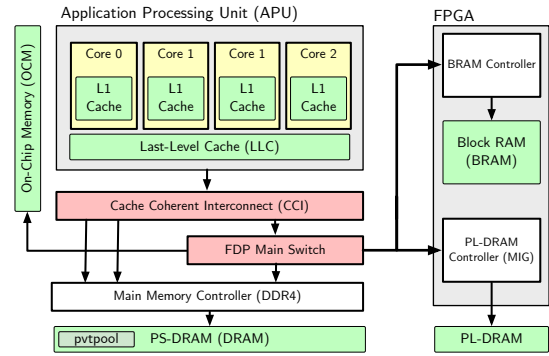


Fig. 3. Overview of evaluation SoC including heterogeneous memory pools.

depicted in Figure 3. The main processor is a 64-bit quad-core ARM Cortex-A53 [15] which uses ARMv8-A [16] ISA and operates at 1.5 GHz. L1 cache comprises 32KB/64KB instruction/data cache with 2-way/4-way set-associativity. The last level cache (L2) is a unified 16-way set associative cache with size of 1MB. The LLC is shared among all cores. The cache line size is 64 bytes for both cache levels.

As shown in Figure 3, our platform features 4 types of memories: (1) the DRAM module that is directly connected to the CPU cluster (*PS-DRAM*), which we refer to as DRAM; (2) the DRAM module that is connected to the programmable logic (PL-DRAM) (3) on-chip scratchpad memory (*OCM*) and, (4) the FPGA-side block random access memory (*BRAM*). In our platform’s Device Tree Blob (DTB), we expose the following memory regions for MemScope’s allocator: 128 KB of OCM, 1 MB OF BRAM, 256 MB of DRAM and PL-DRAM. These sizes represent the slices we carve out for benchmarking; the underlying hardware supports larger capacities.

For the experiments in Section IV-D, we use cache partitioning via page coloring through the Minerva Jailhouse [17]. Cache partitioning allows us to isolate the effects of conflicts on cache sets from the effects of contention on downstream memory modules and shared bus segments. This configuration defines two contiguous intermediate physical address (IPA) ranges. The first IPA range includes all normal memory used by Linux and is mapped by Jailhouse to 12 out of 16 (i.e., 3/4) of the available colors. The second range is mapped to pages using the remaining 4 out of 16 (i.e., 1/4) colors. This range is then exported to MEMSCOPE as a memory pool. As such, only benchmarks with pages allocated from this pool will be able to use the *private* 25% portion of the L2 cache (256 KB). We refer to this pool as the *private cache pool*, namely *pvtpool* in our experiments. From MEMSCOPE’s point of view, this is a distinct heterogeneous memory module.

MEMSCOPE supports configurable iteration counts for workload execution to ensure the statistical stability of the measured performance metrics. In all experiments discussed in this section, we configured this iteration count to 500.

In our results, we use different access strategies. The full list of supported strategies is provided in Table I. We use tuples of the form (a, b) , where a indicates the access strategy employed by the core under observation while b that of a stressor core. For instance, with (r, w) , the core

TABLE I
AVAILABLE ACCESS STRATEGIES IN MEMSCOPE.

Access Pattern	Description
r	sequential reads to benchmark memory read bandwidth
w	sequential writes to benchmark memory write bandwidth
l	data-dependent random reads (pointer chasing) to benchmark latency
s	non-cacheable version of the r benchmark
x	non-cacheable version of the w benchmark
m	non-cacheable version of the l benchmark
y	non-cacheable write-streaming to the memory (no write-allocate)

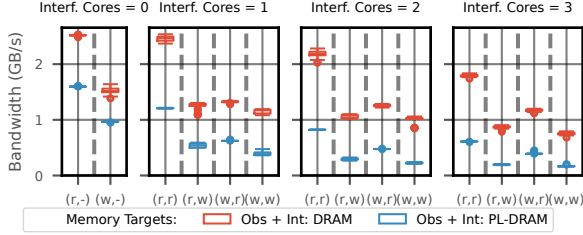


Fig. 4. Homogeneous bandwidth results for DRAM and PL-DRAM under four stress scenarios with buffer size of 4 MB.

under observation performs sequential reads while stressor cores execute sequential writes.

B. Analysis of DRAM Modules

In this subsection, we describe the results of MEMSCOPE’s characterization of the two DRAM memory types in our platform (DRAM and PL-DRAM), in terms of bandwidth, latency, and memory-level parallelism under various scenarios.

We use MEMSCOPE to understand how DRAM and PL-DRAM behavior changes in isolation as operations vary and how they react under different levels of stress. To this end, we test two *homogeneous* setups and two *heterogeneous* setups. In the homogeneous setups—Subsections IV-B(1), IV-B(2), and IV-B(3)—we observe the behavior of the DRAM (resp., PL-DRAM) while the stressors also target the DRAM (resp., PL-DRAM) module. Conversely, in the heterogeneous setups—Section IV-B(4)—we observe the behavior of the DRAM (resp., PL-DRAM) while the stressors target the PL-DRAM (resp., DRAM) module. In these experiments, the buffer size is 4 MB unless otherwise specified.

1) **Homogeneous Bandwidth Analysis:** Figure 4 shows the bandwidth extracted by the observed core from the two DRAM memory types. As expected, it decreases as the number of interfering cores increases. However, this drop is more noticeable in DRAM than in PL-DRAM. The DRAM bandwidth drop becomes noticeable with more than one interfering core in the (r, r) case; it is substantial in the (r, w) case, even with only one stressor core. This is expected, as the cache system follows the write-allocate/write-back (WAWB) policy, meaning that every store resulting in a write miss causes both a memory read and a write-back of some dirty line being evicted. This implicit read in case of write miss can further exacerbate contention effects. Additionally, read operations on the core under observation are synchronous (due to its in-order nature). Thus, pending loads cause pipeline stalls that directly affect the end-to-end execution time and that are amplified if the stressors produce read+write traffic caused by store-heavy access. Conversely, for the DRAM under (w, r) operations, the bandwidth remains relatively stable due to the opposite

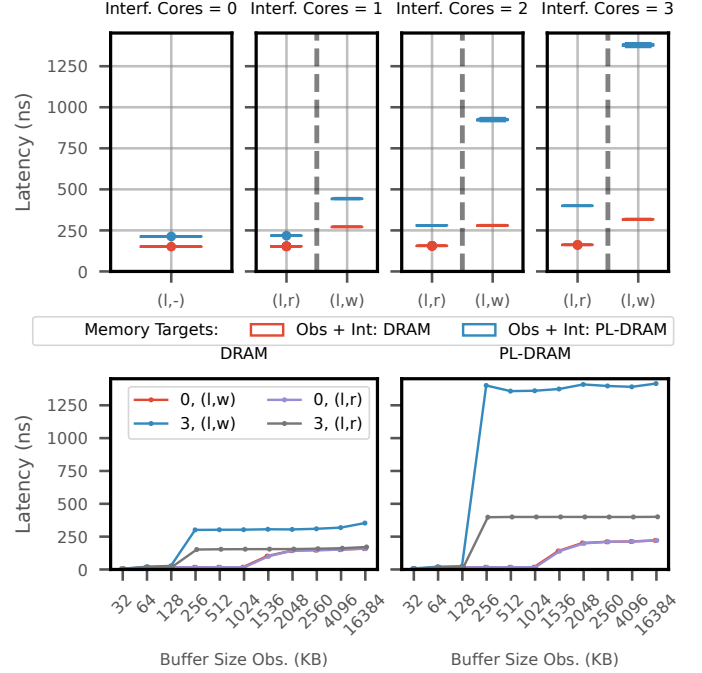


Fig. 5. Homogeneous latency results for DRAM and PL-DRAM under four stress scenarios with buffer size of 4 MB.

effect of the logic discussed. PL-DRAM follows a similar trend, albeit remaining consistently at a lower performance level and with proportionally lower performance degradation. The trend similarity, moreover, highlights how the behavior is characteristic of DRAM technology in spite of substantial differences in clocking, capacity, and manufacturers.

MEMSCOPE allows us to make the following observations: (1) the bandwidth of PL-DRAM is lower than DRAM, as expected, due to its greater distance from the cores and lower clock domain (PL), (2) scenarios exist where a stressed DRAM—e.g., in the (r, w) case—exhibits a bandwidth comparable to that of a non-stressed PL-DRAM, and (3) the stress-induced bandwidth degradation for the DRAM module is proportionally more pronounced compared to PL-DRAM. A heterogeneous memory allocator can leverage these insights.

2) **Homogeneous Latency Analysis:** Figure 5 shows the results of latency analysis using MEMSCOPE using access strategies (l, r) and (l, w) —see Table I. With increasing stress, the latency gap between best- and worst-case scenarios grows. Interestingly, DRAM and PL-DRAM both start from almost the same latency. However, this gap widens as contention worsens. The change in latency for both read and write stress in DRAM remains relatively stable. In contrast, PL-DRAM reacts significantly to the increase in stressors.

The line plots at the bottom of Figure 5 show the measured latency in scenarios with 0 and 3 stressor cores for increasing buffer sizes. In the 0-stressors case, caching effects disappear for buffer sizes above 1 MB; in the 3-stressors case, they disappear for sizes above 256 KB. For DRAM, the latency variation from the best- to the worst-case remains stable at around of 0.3 ns, whereas for PL-DRAM, the latency fluctuates between 1.3 and 1.4 ns.

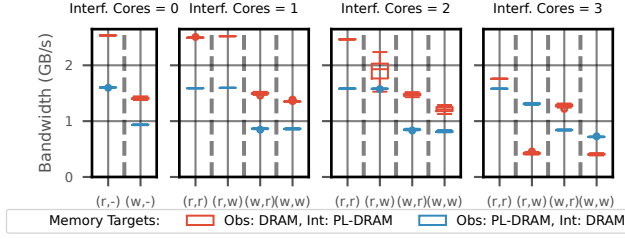


Fig. 6. Heterogeneous bandwidth results for DRAM and PL-DRAM under four stress scenarios with buffer size of 4 MB.

TABLE II
MLP CALCULATION FOR DRAM AND PL-DRAM

	Lat. Operation	BW Operation	Lat.(ns/ T_x)	BW(T_x /ns)	MLP
DRAM	(l,r)	(r,r)	161.89	0.03	4.85
	(l,w)	(r,w)	318.56	0.014	4.45
PL-DRAM	(l,r)	(r,r)	399.49	0.01	3.99
	(l,w)	(r,w)	1386.80	0.003	4.16

3) **MLP Derivation:** Table II displays the measured *Memory-Level Parallelism (MLP)* for DRAM and PL-DRAM. MLP is calculated for both memory types using *Little's Law*, stating that for a system at steady state, the average MLP can be estimated as: Avg. MLP = Avg. Latency $\times$ Avg. Bandwidth. For this analysis, we use the results captured in the worst-case scenarios, where all the interfering cores are executing memory-intensive read/write operations. For bandwidth measurements, we select cases that maximize the throughput. We evaluate the MLP perceived by the core under analysis (access strategy 1), in the case when the other cores perform sequential reads (r) or writes (w). Thus, we pair latency experiments (l, r) with bandwidth experiments (r, r) and (l, w) with (r, w).

We observe comparable values of MLP between the two memory modules in spite of the substantially higher latencies observed under stress for PL-DRAM. This potentially highlights that the bottleneck on the number of outstanding memory transactions lies in the bus infrastructure (CCI, see Figure 3) that is common for transactions targeting either module. However, because PL-DRAM transactions have significantly higher latency, outstanding transactions generally occupy bus-level queue entries for longer. This can reduce the opportunity for transactions targeting faster memory (e.g., DRAM) to progress, effectively throttling its throughput.

This observation motivated the next set of experiments presented in the following section. Our goal is to investigate how mixed access to two memory systems with significant latency disparity respond under stress.

4) **Heterogeneous Bandwidth Analysis:** We present a heterogeneous bandwidth and latency analysis to address the following question: *how does temporal behavior change when the target memory for the core under observation differs from that of the interfering cores?* To explore how MEMSCOPE captures microarchitectural effects arising from the mixed use of two memories with comparable MLP but significantly different latencies, we consider two experiments: (1) The core under observation targets DRAM, while interfering cores target PL-DRAM. This case is labeled as "Obs: DRAM, Int: PL-DRAM" and color-coded in red. (2) The core under

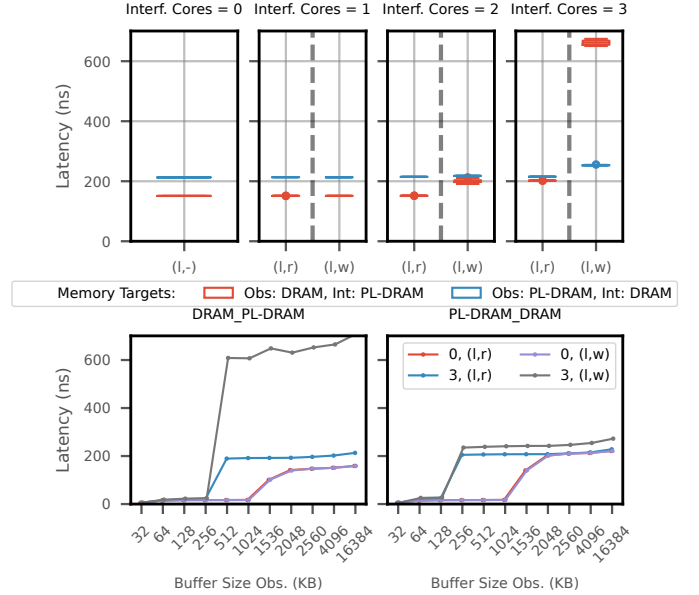


Fig. 7. Heterogeneous latency plots for DRAM and PL-DRAM under four stress scenarios with buffer size of 4 MB

observation targets PL-DRAM, while interfering cores target DRAM, labeled "Obs: PL-DRAM, Int: DRAM," in blue. Figure 6 reports the results.

In the "Obs: DRAM, Int: PL-DRAM" case, DRAM initially outperforms PL-DRAM in isolation. However, as more interfering cores are added, PL-DRAM exhibits more stable performance, with fewer fluctuations compared to DRAM. The results suggest that the degradation is not due to direct contention over DRAM—since bandwidth is measured for DRAM and the interfering cores stress PL-DRAM. Conversely, it suggests a bottleneck elsewhere in the system. At the highest interference level (three interfering cores), the results clearly reflect that saturating the PL-DRAM causes large performance degradation for accesses to the DRAM.

This effect would be counterintuitive without the previously examined MLP and latency analyses. Indeed, even with comparable MLP values, the higher latency of PL-DRAM (under stress) can delay DRAM transactions when their paths overlap in shared bus elements, such as at the level of CCI (Figure 3). The increased latency of pending PL-DRAM transactions causes them to occupy shared bus queue entries longer, thereby reducing availability for DRAM-bound requests.

A similar observation, but in reverse, is presented in Figure 7, where we focus on latency analysis. In the "Obs: DRAM, Int: PL-DRAM" experiment, a noticeable increase in latency is observed when the heterogeneous system becomes congested. This indicates that DRAM, despite its higher standalone bandwidth, is substantially more prone to latency degradation under high-stress mixed memory usage. The line plots (bottom of Figure 7) for the same case clearly highlight this trend. PL-DRAM is not affected by the issue, as shown by the results for the "Obs: PL-DRAM, Int: DRAM" case.

C. Scratchpad Analysis

We showcase how MEMSCOPE can be used to analyze the performance of scratchpad memories available in the system.

These correspond to (1) the On-Chip Memory (OCM) module on the PS side of the SoC and (2) a Block RAM (BRAM) module on the PL side of the SoC—see Figure 3.

In our platform, although the OCM capacity is 256 KB, only 128 KB is reserved for the memory pool, while the BRAM pool is 1 MB. Given the L1 and L2 cache sizes (32 KB and 1 MB, respectively), using cacheable operations would lead to cache hits, misrepresenting actual memory behavior. Therefore, non-cacheable operations are necessary to conduct scratchpad memory analysis. To address this, we leverage the non-cacheable version of MEMSCOPE’s bandwidth (s for reads, and x and y for writes) and latency (m) workloads, as reported in Table I.

As described in Section III, non-cacheable read workloads (s and m) perform a combination of cache line accesses followed by cache clean+invalidations. We always perform 500 iterations in each scenario. Thus, after the first access and invalidation, all the subsequent accesses are ensured to miss in cache. We use two types of non-cacheable write operations. The first, denoted as x , issues store operations followed by cache invalidations. Since the cache policy is WAWB, reads from memory to load cache lines will still occur. Conversely, the y access strategy employs streaming writes, which follow a write-no-allocate policy, bypassing the cache.

1) **Homogeneous Bandwidth Analysis:** Figure 8 presents our measurements for homogeneous bandwidth analysis of OCM and BRAM. As interference increases, OCM bandwidth progressively degrades from the (s, s) case to the (x, y) case. This trend mirrors the behavior discussed in Section IV-B, where read operations are more vulnerable to interference due to the non-blocking nature of writes. When measuring read bandwidth, (s, y) , which employs write streaming, results in the lowest observed bandwidth. BRAM exhibits the same decreasing trend observed for OCM, and its absolute bandwidth remains consistently lower than OCM.

2) **Homogeneous Latency Analysis:** Figure 9 reports latency results for BRAM (red) and OCM (blue) using the m access pattern. With no interference, $(m, -)$ case, OCM outperforms BRAM, showing lower and more stable latency. As interference increases, OCM maintains tighter and lower latency. In contrast, BRAM exhibits higher median latency across most interference, especially under (m, x) and (m, y) . In conclusion, BRAM shows higher sensitivity to interference compared to OCM, making OCM a more reliable choice for the allocation of latency-critical memory pages.

D. Cache Analysis

In the experiments presented in this section, we leverage MEMSCOPE to reproduce the effect of *cache bank contention under hits* previously observed in [18]. We also use this experiment to validate that the measurements obtained through MEMSCOPE match those observable with traditional benchmarks. In particular, we compare MEMSCOPE to the bandwidth benchmark from the IsolBench suite². First,

²<https://github.com/CSL-KU/IsolBench/blob/master/bench/bandwidth.c>

TABLE III
EVENT COUNTS UNDER VARYING INTERFERENCE LEVELS

Event/Interf. cores	Zero	One	Two	Three
CPU_CYCLE	17,131,051	26,228,725	39,834,512	53,836,500
MEM_ACCESS	2,049,051	3,764,331	3,760,759	3,748,782
L2D_CACHE	3,855,710	3,764,331	3,760,759	3,748,782
L2D_CACHE_REFILL	5,182	204	1,748	5,591
Cache Hit Rate	99.87%	99.99%	99.95%	99.85%
Cycles/Access	4.44	6.97	10.59	14.36

Figure 10 compares two DRAM bandwidth measurement experiments: one using IsolBench (color-coded in red), and the other using MEMSCOPE (color-coded in blue). In both cases, the buffer size per core is set to 256 KB: larger than L1 but small enough to fit within the LLC, ensuring that all accesses are hits. Thus, both experiments target the same memory module and follow equivalent configurations. The very close match in the measurements obtained using the two benchmarking approaches serves as validation that what is observed with MEMSCOPE is indeed in line with established memory performance benchmark measurement toolkits, justifying further analysis relying solely on MEMSCOPE.

1) **Bank contention under cache hits:** Having ascertained that the cache-hit performance drop under stress identified by MEMSCOPE is repeatable, we conduct a further experiment to verify that indeed the source of the performance degradation observed in Figure 10 can be attributed to the problem of cache bank contention, as previously studied in [18].

To this end, we leveraged the integrated support for performance counter sampling in MEMSCOPE. We sampled the counters on the core under observation, focusing on four key metrics listed in Table III: CPU cycles (CPU_CYCLE), data memory accesses (MEM_ACCESS), L2 data cache accesses (L2D_CACHE), and L2 data cache refills (L2D_CACHE_REFILL). The results confirmed our hypothesis: cache hit rates ($>99.8\%$) aligned with expectations. However, the number of CPU cycles per cache access increased notably ($3.23\times$). As such, we conclude that the effect arises from cache bank-level contention on the hit path.

2) **Bank contention under hits, with cache partitioning:** Since cache bank contention on the hit path [18] is unaffected by cache partitioning, we postulate that the same results should be obtainable with MEMSCOPE. Indeed, while cache partitioning divides the cache space, it does not deconflict the banks, so contention at the bank level remains. We use MEMSCOPE to reproduce this effect for the first time on a platform featuring in-order Cortex-A53 cores with a single-bank LLC, while it was previously observed on out-of-order Cortex-A72, Cortex-A52, and Xuantie C910 [18], [19]. First, as mentioned in Section IV-A, we leverage the Jailhouse partitioning hypervisor to export a 25% private L2 cache reservation as a memory pool, namely `pvtpool`. Since this is yet another pool, we can utilize the full array of benchmarks available in MEMSCOPE. For this experiment, we focus on bandwidth behavior.

Next, we use a similar setup as per Figure 10 in Figure 11, i.e., where all the cores hit in L2 cache (256 KB buffers). However, in these experiments, we consider the cases in which

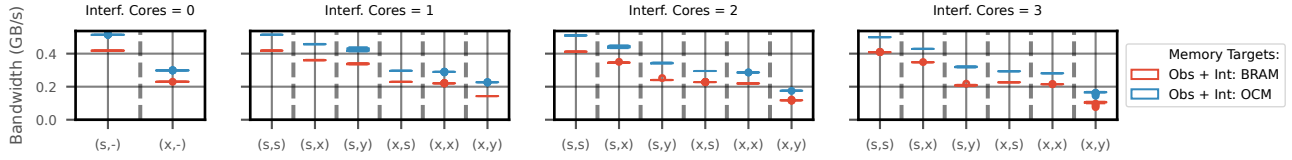


Fig. 8. Homogeneous bandwidth results for OCM and BRAM under four stress scenarios with 32KB buffer.

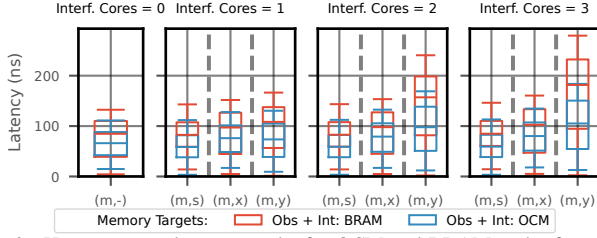


Fig. 9. Homogeneous latency results for OCM and BRAM under four stress scenarios with 32KB buffer.

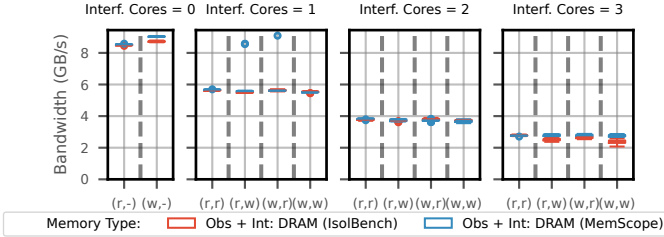


Fig. 10. LLC bandwidth measurement for buffer size 256KB using the IsolBench suite (color-coded in red) and MEMSCOPE (color-coded in blue).

partitioning is disabled (red) vs. enabled (blue). In the latter case, the observed core strictly allocates from the private cache partition (`pvtpool`). As expected, due to hit-path bank contention, partitioning is ineffective in mitigating contention.

3) **Bank contention under miss, with cache partitioning:** We conducted additional experiments to understand the conditions under which the system benefits from cache partitioning. Figure 12 presents such a case. In these experiments, we consider two cases. In the first case (red), all the cores access 256 KB from the shared cache partition—which is 3/4 of the L2, thus 768 KB. Thus, the observed core suffers inter-core evictions. In the second case (blue), the observed core accesses 256 KB mapping to the private cache partition (`pvtpool`), hitting in cache; all the stressor cores access 4 MB from the

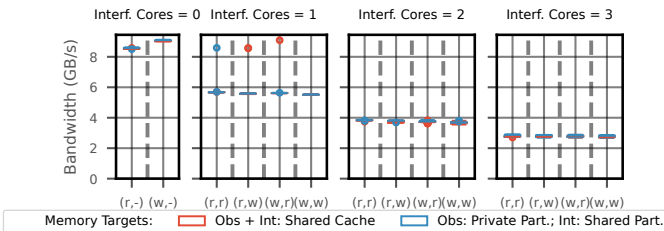


Fig. 11. Bandwidth measurement, for buffer size 256 KB, with and without cache partitioning.

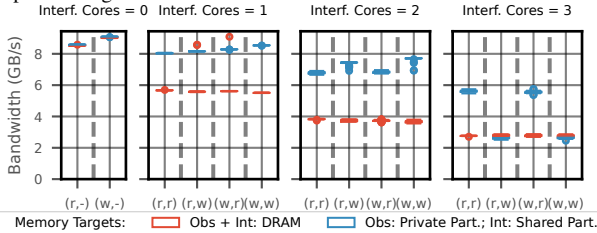


Fig. 12. Bandwidth measurement with all cores accessing 256 KB in shared cache partition (red) vs. observed core accessing 256 KB from private cache partition and all interfering cores accessing 4 MB from shared partition (blue).

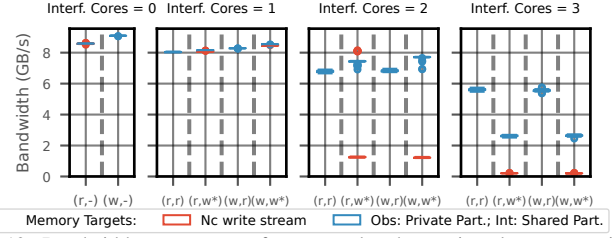


Fig. 13. Bandwidth measurement for core under observation always accessing 256 KB in private cache partition. Interfering cores access 4 MB from shared cache partition w/ normal reads/writes (blue) vs. write-streaming (red).

shared cache partition, missing in cache. The plot shows that cache partitioning is effective for most types of interfering workloads. The only exceptions are the (r, w) and (w, w) cases, where miss-path cache bank contention occurs.

MEMSCOPE allows us to push the effects of miss-path cache bank contention to the limit. To test this, in Figure 13, we run an experiment where the core under observation always accesses 256 KB from the private cache partition while interfering cores use normal reads/writes (blue) to access 4 MB from the shared cache partition—this is identical to the blue case in Figure 12. Next, we compare it to the case (red) where no changes are made to the observed core, while the interfering cores still access 4 MB from the shared cache partition, but they do so using non-cacheable write-streaming operations— y access strategy, see Table I. For the non-cacheable write-stream experiment, we evaluated the (r, y) and (w, y) combinations. Since some experiments included both w (normal cacheable write) and y (non-cacheable write stream) operations, we use the w^* notation in the plot. This corresponds to w for the case where stressors use normal writes (blue) and to y otherwise (red).

The results clearly show that while the measured bandwidth is identical in the case of one stressor core, drastic performance degradation is caused by streaming writes with two or more active interfering cores, in spite of cache partitioning. The very high performance loss (about $40\times$) is in line with similar results from [20] also obtained on Cortex-A53 platforms.

E. Management of Real-Time Applications using MEMSCOPE

In this section, we investigate how applications can practically leverage the insights captured by MEMSCOPE. Typically, real-time management of applications accessing shared memory subsystems relies on memory bandwidth regulation. In this subsection, we present a new dimension of management enabled by the insights captured through MEMSCOPE and its ability to characterize the full heterogeneous memory subsystem. In addition to traditional bandwidth regulation, it becomes possible to make informed decisions about which memory type should be used by a given application. Understanding the memory characteristics and predicting their behavior under stress can significantly aid this determination.

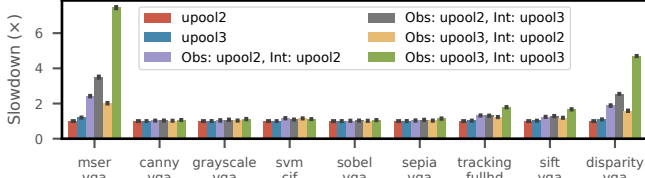


Fig. 14. Heterogeneous memory management in real-world applications.

TABLE IV

COMPARISON WITH OTHER MEMORY BENCHMARKING TOOLS

Prior Work	Open Source	Multi Core	Heterogeneous Memory	Kernel Mode	Performance Counters	Supported Architectures
Intel MLC [13]	✓	✓	✓	✓	✓	x86
Isolbench [25]	✓	✓	✓	✓	✓	x86/Arm
Nanoench [12]	✓	✓	✓	✓	✓	x86
Heimdall [26]	✓	✓	✓	✓	✓	x86
LENS [27]	✓	✓	✓	✓	✓	x86
tinymembench [28]	✓	✓	✓	✓	✓	x86/Arm
MEMSCOPE	✓	✓	✓	✓	✓	Arm

We perform our analysis on benchmarks from the San Diego Vision Benchmark Suite (SD-VBS) [21] and the Image Filters from RT-Bench [22]. RT-Bench provides the ability to map the benchmark’s heap to any of the pools exported by MEMSCOPE to user space (*upool*), as described in Section III. We investigate how their end-to-end runtime varies by changing where the heap is mapped and present the results in Figure 14.

We focus on *upool2* and *upool3*, which correspond to DRAM and PL-DRAM, respectively. The *x*-axis reports the name of the benchmark alongside its input size. The *y*-axis depicts the slowdown, with each job’s duration normalized to the baseline (1st bar) defined as the case where the application runs in isolation and allocates solely from *upool2* (normal DRAM). The 2nd bar corresponds to the in-isolation run with the heap allocated in *upool3* (PL-DRAM).

In the 3rd, 4th, 5th, and 6th bars in each cluster, the legend reports the *upool* used to allocate the heap of the observed application, while write-heavy interference from 3 stressors is introduced targeting *upool2* and *upool3*, as per legend.

The performance macro-trends observed in Figure 14 align with the insights captured by MEMSCOPE. Indeed, although it may initially seem counterintuitive without MEMSCOPE’s guidance, allocating pages of the target application from DRAM (via *upool2*) while stressors target PL-DRAM (via *upool3*) results in higher slowdowns compared to the inverse setup. This is true across all the benchmarks and especially noticeable in benchmarks like *mser* and *disparity*. We postulate that the variation in sensitivity observed across the benchmarks is related to differences in their access patterns and memory-boundness. Said differences have been studied using workload profiling tools [9]–[11] and examining sensitivity to memory subsystem performance [23], [24].

V. RELATED WORK

Performance characterization is crucial for any heterogeneous system. When a system features resource heterogeneity (in compute and/or memory), it must continuously decide how to optimally utilize these diverse resources for varying compute demands. Making such decisions is only possible with a thorough understanding of the performance characteristics of each individual resource. As a result, performance characterization has been the focus of many studies [26], [29],

[30]. In particular, the performance of memory subsystems has received significant attention [26], [30]–[34]. Most of these studies, however, focus on either cache behavior [32]–[34] or a single memory technology [25], [26], [30], [31], often in general-purpose, high-performance settings. Furthermore, the majority are implemented in user space, making them subject to the limitations outlined in Section II. In contrast, Memscope offers a precise, extensible, kernel-level, open-source framework specifically designed for heterogeneous memory systems in embedded real-time environments. Table IV shows a high-level comparison of MEMSCOPE with closely related memory benchmarking tools. In the rest of this section, we survey works in the broader area of memory characterization.

a) Characterizing Caches: Several prior studies have proposed microbenchmark techniques to determine cache hierarchy parameters, such as cache size, associativity, block size, and latency [32]–[40]. These studies are performed either to guide performance optimization [32]–[34], or for performing cache side-channel attacks [40], [41]. Most of these work assume a constant penalty for accesses that miss the cache and thus need to go to a single-technology main memory. While MEMSCOPE’s microbenchmarks also often need to consider caches—mostly to bypass them and reach to the main memory, Memscope’s goal is different. It provides a benchmarking framework to precisely characterize a heterogeneous memory system beyond just cache properties.

b) Characterizing Single Memory Technology: Prior work also extensively studied performance properties of a single memory technology as the main memory. DRAM is perhaps the most studied one [31], [42]–[45]. SoftMC [31] offers an open-source FPGA-based benchmarking platform that can test DRAM memory modules through a DDR interface, by directly sending DDR commands to the modules and measuring the response time. More recently DRAM Bender [42] builds on top of SoftMC and provides users the ability to write DRAM-based tests in high-level programming languages such as python. There are also other benchmarking studies that try to determine undocumented DRAM properties such as the refresh mechanism [43], DRAM row buffer [44], and DRAM address to row mappings [44]–[46]. Similarly there many studies to understand low-level device-level characteristics of other memory technologies such as NVM [27], [30], [47], HBM [48]–[51], and PIM [52]. In contrast to these studies, MEMSCOPE does not target only one memory technology, but focuses on understanding the entire heterogeneous memory system, including how different memories affect each other.

c) Characterizing Heterogeneous Memory: The majority of prior work on characterizing heterogeneous memory systems has focused on general-purpose, high-performance computing. Modern high-performance multicore servers typically feature a non-uniform memory access (NUMA) design, where clusters of cores share a single memory controller, and nodes are interconnected via high-speed links. In such systems, any core can access memory attached to the entire system, but with non-uniform latency, as the access time depends on the memory location relative to the requesting core.

This introduces challenges similar to those in heterogeneous memory systems. Several studies [53], [54] have characterized NUMA performance to optimize overall system efficiency. The introduction of persistent memory modules (such as Intel’s Optane) has added another layer of heterogeneity in high-performance computing, and prior work has explored their performance characteristics [27], [30], [47], [55]. More recently, CXL (Compute Express Link) has emerged as a cache-coherent interconnect built on top of PCIe, allowing systems to add memory modules to the CXL fabric, introducing yet another form of heterogeneity. The performance of CXL-based memory has been the focus of several recent studies [26], [56].

RAMify [57] is a complementary framework that generates fine-grained, customizable memory-centric workloads for early-stage design space exploration, often in simulation. Its targeted workload generation, including worst-case-oriented real-time stress patterns, can be used to design hardware characterization experiments for MEMSCOPE on physical platforms. Moreover, the characterization of heterogeneous memory, as in our work, can inform frameworks that optimize data placement. For example, MOCA [58] targets user-space memory objects and KLOCs [59] targets kernel objects, allocating them to faster/slower memory tiers.

d) Generic Benchmarking Frameworks: Prior work has also proposed generic microbenchmarking tools that allow users to infer performance characteristics of user-provided code, usually through measuring performance counters. Linux perf [60] allows users to measure performance counters for a particular executable. Agner tool [61] gives users more control by allowing measurement for a particular part of the code. Similarly, Nanobench [12] also allows users to read performance counters of a microbenchmark written for x86 and runs in kernel mode. However, unlike MEMSCOPE, nanobench does not provide multi-core microbenchmarks for heterogeneous memory characterization and only supports x86.

VI. PROFILE-DRIVEN RTOS MEMORY ALLOCATION

The coexistence of multiple memory technologies with different latency, bandwidth, and capacity characteristics introduces *technological heterogeneity*, while the non-uniform performance impact of different application pages creates *usage heterogeneity*, as discussed in Section II. Together, these two factors introduce a gap between the theoretical capabilities of modern heterogeneous memory subsystems and the ability of modern real-time operating systems (RTOS) to leverage said capabilities. This gap presents both a challenge and an opportunity, i.e., devising practical approaches to best map application pages onto heterogeneous memory resources.

Indeed, traditional RTOS memory allocation strategies are designed without considering the aforementioned sources of heterogeneity. Therefore, little to no customization is exerted over the physical memory allocation of individual application pages. This results in the inability to optimize memory management by leveraging different memory types and technologies to guarantee access latency constraints. On the contrary, by capturing and exploiting awareness of such heterogeneity,

it is possible to propose and study a number of solutions to the heterogeneous memory allocation problem, to achieve better temporal predictability and/or performance tuning in time-sensitive applications.

Profiling Usage Heterogeneity. The first step towards a profile-driven heterogeneous memory subsystem involves the extraction of per-page *importance* metrics to understand usage heterogeneity for the application(s) under analysis. In this context, given a profiled memory page, the notion of importance must capture a quantification of the end-to-end temporal impact induced by the latency to access the memory blocks contained in the page. Per-page importance metrics can be expressed in *absolute* terms, for instance, by computing the worst-case end-to-end runtime improvement/degradation caused by allocating the page under analysis from a memory resource with a given access latency. Importance metrics can also be expressed in *relative* terms. In this case, only a ranking of pages is created, with the insight that prioritizing allocation in low-latency memory following said ranking maximizes the impact on the end-to-end execution time. While more research is required to create comprehensive page profiling toolkits, existing literature in this space has already explored the extraction of memory profiles for real-time applications. Most prominently, the BBProf [11] toolkit enables the construction of relative importance rankings in the form of binary profiles.

Profiling Technological Heterogeneity. The second milestone in truly profile-driven OS memory management subsystems consists of acquiring and retaining information about the performance characteristics of heterogeneous memory components. In this sense, MEMSCOPE provides a comprehensive solution to construct said characterization. In practice, however, driving this characterization using expert insights (as carried out in Section IV) might be costly on each new hardware platform. Thus, we envision that it might be worth engineering semi-automatic characterization toolkits built atop the proposed MEMSCOPE. Note that in some cases, the temporal behavior of memory components might change even on the same hardware platform, e.g., due to a change in the boot parameters concerning power management settings and/or clock tuning. Semi-automatic characterization is therefore attractive because it could be used offline to construct the initial parameters, but also at boot-time to validate the current memory characteristics. After proper characterization has been obtained (and validated), it must remain available in the OS across boot cycles and be readily available as lookup information when performing page allocations. Multiple approaches can be used to this end. As mentioned in Section III, (RT)OS’s use hardware description files (e.g., device trees) to understand the hardware configuration at hand. Once acquired, memory characterization profiles could be embedded within the same hardware description files.

Once both per-application page profiles and memory resource profiles have been acquired, how they shall be used to perform profile-driven memory management depends on the class of system under consideration. Thus, we hereby distinguish between designs that apply to (1) microcontroller-

grade RTOS's where physical memory is statically provisioned at boot time vs. (2) rich microprocessor-grade, Linux-like RTOS or Linux-based (soft) real-time solutions, where physical memory is allocated via demand paging.

Profile-driven Boot-time Memory Allocation. In deterministic allocation settings where all memory is statically committed at boot, the page-to-resource assignment can be formulated as an offline optimization problem, akin to what was proposed in [62]. An optimal static mapping can then be computed based on application page profiles and MEMSCOPE-provided memory characterization models. The main advantage of this solution consists of the ability to have a global view of the memory demand of all the tasks that will be active in the system and attempt to achieve a global optimum. Additionally, the allocation strategy is consistently applied at every boot, ensuring predictability across reset cycles. The main drawbacks are (1) the inherent complexity due to the high dimensionality of the problem, and (2) a costly re-evaluation of the full allocation strategy upon any update to the taskset.

Profile-driven Runtime Memory Allocation. In systems that rely on demand paging or support open workloads, new tasks can be introduced at runtime. Moreover, physical memory may not be allocated immediately, but only when a page fault occurs, as is often the case in many soft real-time systems. Granted, it is customary to pre-fault memory pages at task creation to improve predictability. Pre-faulting a memory page forces the population of a page table entry early on (e.g., at process initialization) instead of performing it after suffering a page fault when the page is accessed for the first time (a.k.a., demand paging). This reduces the number of page faults that occur when a program accesses a virtual memory page not yet mapped to physical memory. In the worst case, page fault handling might require interaction with I/O devices (e.g., fetching data from disk), resulting in slower and/or less predictable execution. In classes of systems that primarily rely on demand paging, a more practical approach to enact profile-driven heterogeneous memory allocation consists of leveraging heuristics. Indeed, global/static memory provisioning might overly constrain the system, and relying on heuristics that use profiling information to guide memory placement decisions as they occur is preferred.

A key requirement for these heuristics is that they remain lightweight because their runtime will directly contribute to the page fault handling latency. As such, any added overhead might directly impact the task's WCET. This is a lesser concern if real-time tasks are engineered to systematically pre-fault the required memory at initialization time. But even then, implementing long-running routines inside a kernel-level allocator must be avoided to prevent blocking of other user-space real-time applications.

Ultimately, a hybrid approach might be the most desirable in this context: more complex and optimal policies can be employed at process creation for pre-faulted pages, while lightweight heuristics are leveraged for runtime demand paging. Balancing the use of the two approaches effectively allows trading off slightly longer process startup time for better

allocation decisions and less pessimistic WCET.

Integration Roadmap. As it emerges from the discussion above, while MEMSCOPE constitutes a fundamental building block towards OS-level profile-driven heterogeneous memory management, dedicated research efforts are still necessary to this end. Exploration of the outlined next steps constitutes part of our ongoing research roadmap. In particular, our efforts are currently directed towards (1) the integration of profiling information for both memory resources and application pages in a microcontroller-grade RTOS; and (2) the design of mid-complexity and low-complexity heuristics that can be employed in microprocessor-grade RTOS's to handle pre-faulted pages and demand paging requests, respectively.

VII. CONCLUSION AND FUTURE RESEARCH AVENUES

MEMSCOPE is a novel kernel-level memory benchmarking framework designed and implemented to characterize the temporal behavior of heterogeneous memory subsystems, particularly in real-time embedded systems. Implemented fully in kernel space, it leverages privileged APIs for fine-grained control of core execution, physical memory allocation, and cache states. MEMSCOPE includes an extensible benchmark library not only for measuring bandwidth and latency but also for observing the relevant microarchitectural behaviors and events. MEMSCOPE offers several new insights into heterogeneous memory performance and enables counter-intuitive yet effective memory management decisions for real-time tasks to reduce contention sensitivity.

Our immediate research roadmap already envisions the use of MEMSCOPE as a crucial building block to design and implement OS-level profile-driven heterogeneous memory management geared toward predictability via careful resource provisioning. In addition, we anticipate that MEMSCOPE could be an invaluable framework in a number of other research domains. In system security research, MEMSCOPE can be used to investigate denial-of-service and timing-based side-channel attacks that exploit memory contention in heterogeneous systems. Furthermore, modern compilers can leverage MEMSCOPE's memory performance insights to determine the optimal placement of performance-critical data structures and routines in high-bandwidth, low-latency memory. Additionally, MEMSCOPE allows system designers to evaluate hypothetical scenarios, like adding heterogeneous memory modules (e.g., FPGA-side BRAM), to see how they might benefit target workloads. MEMSCOPE's design is agnostic to specific memory types; thus, it can be used to evaluate the benefits of integrating emerging technologies such as HBM, LPDDR5, NVM, or CXL-attached memory. Last but not least, MEMSCOPE's insights can aid in refining WCET analysis. For measurement-based approaches, MEMSCOPE can help discover and exercise counterintuitive memory behaviors by reproducing extreme stressing conditions. For static analysis, MEMSCOPE-derived insights can refine the microarchitectural models used for WCET estimation on complex multi-core systems.

ACKNOWLEDGMENTS

This research was supported by the National Science Foundation (NSF) under grant number CSR-2238476 and by a gift from Ampere Computing. Different co-authors used Grammarly and Chat-GPT solely with the intention of assisting in grammatical correction and enhancement.

REFERENCES

- [1] A. Melo, J. Carretero, P. Stenstrom, S. Ranka, and E. Ayguade, "Trends on heterogeneous and innovative hardware and software systems," *Journal of Parallel and Distributed Computing*, vol. 133, pp. 362–364, 2019.
- [2] The Linux Kernel Community, *Linux Kernel Documentation: Remote Processor Framework*. The Linux Foundation, 2024. <https://docs.kernel.org/staging/remoteproc.html>.
- [3] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.
- [4] M. Hassan, "On the off-chip memory latency of real-time systems: Is DDR DRAM really the best option?," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 495–505, 2018.
- [5] K. Asifuzzaman, M. Abuelala, M. Hassan, and F. J. Cazorla, "Demystifying the characteristics of high bandwidth memory for real-time systems," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, p. 1–9, IEEE Press, 2021.
- [6] M. Bazzaz, A. Hoseinghorban, and A. Ejlali, "Fast and predictable non-volatile data memory for real-time embedded systems," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 359–371, 2021.
- [7] D. Oehlert, A. Luppold, and H. Falk, "Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)* (M. Bertogna, ed.), vol. 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 1:1–1:22, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.
- [8] S. Wasly and R. Pellizzoni, "A dynamic scratchpad memory unit for predictable real-time embedded systems," in *2013 25th Euromicro Conference on Real-Time Systems*, pp. 183–192, 2013.
- [9] T. Janjusic and K. Kavi, "Gleipnir: a memory profiling and tracing tool," *SIGARCH Comput. Archit. News*, vol. 41, p. 8–12, Dec. 2013.
- [10] H. Brais and P. R. Panda, "Alleria: An advanced memory access profiling framework," *ACM Trans. Embed. Comput. Syst.*, vol. 18, Oct. 2019.
- [11] G. Ghaemi, D. Tarapore, and R. Mancuso, "Governing with Insights: Towards Profile-Driven Cache Management of Black-Box Applications," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)* (B. B. Brandenburg, ed.), vol. 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 4:1–4:25, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.
- [12] A. Abel and J. Reineke, "nanobench: A low-overhead tool for running microbenchmarks on x86 systems," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 34–46, IEEE, 2020.
- [13] Intel, "Intel® memory latency checker v3.11b," <https://www.intel.com/content/www/us/en/developer/articles/tool/intel-memory-latency-checker.html>, 2014.
- [14] Xilinx, Inc., "Zynq UltraScale+ MPSoC data sheet: Overview (v1.8)," https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf, 2019.
- [15] ARM Holdings, "Cortex-A53 MPCore technical reference manual (r0p4)," <https://developer.arm.com/documentation/ddi0500/j/>, 2018.
- [16] A. Holdings, "Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile (version G.a)," 2011.
- [17] Minervasys, "Jailhouse," <https://github.com/Minervasys/jailhouse>, 2023.
- [18] M. Bechtel and H. Yun, "Cache bank-aware denial-of-service attacks on multicore ARM processors," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 198–208, 2023.
- [19] C. Sullivan, A. Manley, M. Alian, and H. Yun, "Per-Bank Bandwidth Regulation of Shared Last-Level Cache for Real-Time Systems," in *2024 IEEE Real-Time Systems Symposium (RTSS)*, (Los Alamitos, CA, USA), pp. 336–348, IEEE Computer Society, Dec. 2024.
- [20] M. Bechtel and H. Yun, "Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (Los Alamitos, CA, USA), pp. 357–367, IEEE Computer Society, Apr. 2019.
- [21] 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)*, pp. 55–64, Oct 2009.
- [22] M. Nicoletta, 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, RTNS 2022*, (New York, NY, USA), p. 184–195, Association for Computing Machinery, 2022.
- [23] Z. Xu, Y. Chon, Y. Su, Z. Tan, S. Apostolakis, S. Campanoni, and D. I. August, "PROMPT: A fast and extensible memory profiling framework," *Proc. ACM Program. Lang.*, vol. 8, Apr. 2024.
- [24] J. Ren, D. Xu, J. Ryu, K. Shin, D. Kim, and D. Li, "MTM: Rethinking memory profiling and migration for multi-tiered large memory," in *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24*, (New York, NY, USA), p. 803–817, Association for Computing Machinery, 2024.
- [25] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 1–12, 2016.
- [26] Z. Wang, S. Mahar, L. Li, J. Park, J. Kim, T. Michailidis, Y. Pan, T. Rosing, D. Tullsen, S. Swanson, et al., "The hitchhiker's guide to programming and optimizing cxl-based heterogeneous systems," *arXiv preprint arXiv:2411.02814*, 2024.
- [27] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 496–508, IEEE, 2020.
- [28] S. Siamashka, "Simple benchmark for memory throughput and latency resources," <https://github.com/ssvb/tinymembench>, 2016.
- [29] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter FPGAs for software programmers through microbenchmarking," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '21*, (New York, NY, USA), p. 105–115, Association for Computing Machinery, 2021.
- [30] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, et al., "Basic performance measurements of the intel optane DC persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.
- [31] H. Hassan, N. Vijaykumar, S. Khan, S. Ghose, K. Chang, G. Pekhimenko, D. Lee, O. Ergin, and O. Mutlu, "SoftMC: A flexible and practical open-source infrastructure for enabling experimental DRAM studies," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 241–252, 2017.
- [32] R. Saavedra and A. Smith, "Measuring cache and TLB performance and their effect on benchmark runtimes," *IEEE Transactions on Computers*, vol. 44, no. 10, pp. 1223–1235, 1995.
- [33] C. Thomborson and Y. Yu, "Measuring data cache and TLB parameters under linux," in *Proceedings of the symposium on Performance Evaluation of Computer and Telecommunication Systems*, pp. 383–390, 2000.
- [34] C. L. Coleman and J. W. Davidson, "Automatic memory hierarchy characterization," in *ISPASS*, pp. 103–110, 2001.
- [35] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You, "Accurate cache and TLB characterization using hardware counters," in *Computational Science-ICCS 2004: 4th International Conference, Kraków, Poland, June 6-9, 2004, Proceedings, Part III 4*, pp. 432–439, Springer, 2004.
- [36] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 181–192, 2005.
- [37] K. Yotov, S. Jackson, T. Steele, K. Pingali, and P. Stodghill, "Automatic measurement of instruction cache capacity," in *International Workshop on Languages and Compilers for Parallel Computing*, pp. 230–243, Springer, 2005.
- [38] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multi-

- processor system,” in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pp. 261–270, IEEE, 2009.
- [39] A. Abel and J. Reineke, “Measurement-based modeling of the cache replacement policy,” in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 65–74, IEEE, 2013.
- [40] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, “Reverse engineering intel last-level cache complex addressing using performance counters,” in *Research in Attacks, Intrusions, and Defenses: 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015. Proceedings 18*, pp. 48–65, Springer, 2015.
- [41] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *2015 IEEE symposium on security and privacy*, pp. 605–622, IEEE, 2015.
- [42] A. Olgun, H. Hassan, A. G. Yağlıkçı, Y. C. Tuğrul, L. Orosa, H. Luo, M. Patel, O. Ergin, and O. Mutlu, “DRAM bender: An extensible and versatile FPGA-based infrastructure to easily test state-of-the-art DRAM chips,” *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 42, p. 5098–5112, Dec. 2023.
- [43] H. Hassan, Y. C. Tuğrul, J. S. Kim, V. van der Veen, K. Razavi, and O. Mutlu, “Uncovering In-DRAM rowhammer protection mechanisms: a new methodology, custom rowhammer patterns, and implications,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO ’21, (New York, NY, USA), p. 1198–1213, Association for Computing Machinery, 2021.
- [44] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, “DRAMA: exploiting DRAM addressing for cross-cpu attacks,” in *Proceedings of the 25th USENIX Conference on Security Symposium, SEC’16, (USA)*, p. 565–581, USENIX Association, 2016.
- [45] C. Helm, S. Akiyama, and K. Taura, “Reliable reverse engineering of intel DRAM addressing using performance counters,” in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 1–8, 2020.
- [46] A. Barengi, L. Breveglieri, N. Izzo, and G. Pelosi, “Software-only reverse engineering of physical DRAM mappings for rowhammer attacks,” in *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pp. 19–24, IEEE, 2018.
- [47] Z. Wang, M. Taram, D. Moghimi, S. Swanson, D. Tullsen, and J. Zhao, “NVLeak: Off-Chip Side-Channel attacks via Non-Volatile memory systems,” in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 6771–6788, USENIX Association, Aug. 2023.
- [48] M. Zhu, Y. Zhuo, C. Wang, W. Chen, and Y. Xie, “Performance evaluation and optimization of HBM-enabled GPU for data-intensive applications,” in *Proceedings of the Conference on Design, Automation & Test in Europe, DATE ’17, (Leuven, BEL)*, p. 1245–1248, European Design and Automation Association, 2017.
- [49] Z. Wang, H. Huang, J. Zhang, and G. Alonso, “Shuhai: Benchmarking high bandwidth memory on FPGAS,” in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 111–119, 2020.
- [50] I. B. Peng, R. Gioiosa, G. Kestor, J. S. Vetter, P. Cicotti, E. Laure, and S. Markidis, “Characterizing the performance benefit of hybrid memory system for HPC applications,” *Parallel Computing*, vol. 76, pp. 57–69, 2018.
- [51] K. Asifuzzaman, M. Abuelala, M. Hassan, and F. J. Cazorla, “Demystifying the characteristics of high bandwidth memory for real-time systems,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2021.
- [52] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, “Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system,” *IEEE Access*, vol. 10, pp. 52565–52608, 2022.
- [53] R. Entezari-Maleki, Y. Cho, and B. Egger, “Evaluation of memory performance in NUMA architectures using stochastic reward nets,” *Journal of Parallel and Distributed Computing*, vol. 144, pp. 172–188, 2020.
- [54] R. Geist and J. Westall, “Performance and availability evaluation of NUMA architectures,” in *Proceedings of IEEE International Computer Performance and Dependability Symposium*, pp. 271–280, 1996.
- [55] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, “Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules,” in *Proceedings of the International Symposium on Memory Systems, MEMSYS ’19, (New York, NY, USA)*, p. 288–303, Association for Computing Machinery, 2019.
- [56] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim, “Demystifying CXL memory with genuine CXL-ready systems and devices,” in *56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO ’23*, p. 105–121, ACM, Oct. 2023.
- [57] M. Abuelala and M. Hassan, *A Framework for Explainable, Comprehensive, and Customizable Memory-Centric Workloads*. New York, NY, USA: Association for Computing Machinery, 2025.
- [58] A. Narayan, T. Zhang, S. Aga, S. Narayanasamy, and A. Coskun, “MOCA: Memory object classification and allocation in heterogeneous memory systems,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 326–335, 2018.
- [59] S. Kannan, Y. Ren, and A. Bhattacharjee, “KLOCs: kernel-level object contexts for heterogeneous memory systems,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’21, (New York, NY, USA)*, p. 65–78, Association for Computing Machinery, 2021.
- [60] A. C. De Melo, “The new linux’perf’tools,” in *Slides from Linux Kongress*, vol. 18, 2010.
- [61] A. Fog, “Test programs for measuring clock cycles and performance monitoring,” <https://agner.org/optimize>, 2025.
- [62] A. Druetto, E. Bini, A. Grosso, S. Puri, S. Bacci, M. Di Natale, and F. Paladino, “Task and memory mapping of large size embedded applications over NUMA architecture,” in *Proceedings of the 31st International Conference on Real-Time Networks and Systems, RTNS ’23, (New York, NY, USA)*, p. 166–176, Association for Computing Machinery, 2023.