Learning Memory-Contention Timing Models With Automated Platform Profiling

Andrea Stevanato*†, Matteo Zini*, Alessandro Biondi*, Bruno Morelli†, and Alessandro Biasci†
*Scuola Superiore Sant'Anna, Pisa, Italy, †Evidence s.r.l., Pisa, Italy

***Abstract*—Commercial Off-The-Shelf (COTS) multi-core platforms are often used to enable the execution of mixed-criticality real-time applications. In these systems, the memory subsystem is one of the most notable sources of interference and unpredictability, with the Memory Controller (MC) being a key component orchestrating the data flow between processing units and main memory. The worst-case response times of real-time tasks is indeed particularly affected by memory contention and, in turn, by the MC behavior as well.**

This paper presents FrATM^2, a Framework to Automatically learn the Timing Models of the Memory subsystem. The framework automatically generates and executes micro-benchmarks on bare-metal hardware to profile the platform behavior in a large number of memory-contention scenarios. After aggregating and filtering the collected measurements, FrATM^2 trains MC models to bound memory-related interference. The MC models can be used to enable response-time analysis. The framework was evaluated on an AMD/Xilinx Ultrascale+ SoC, collecting gigabytes of raw experimental data by testing tents of thousands of contention scenarios.

I. Introduction

Commercial Off-The-Shelf (COTS) embedded platforms are widely adopted in multiple industrial domains, including automotive, industrial automation, robotics, and medical devices. These systems often require execution of mixed-criticality real-time applications [1], whose performance is known to be significantly affected by memory-contention delays [2], [3], especially when exchanging large amounts of data [4].

In these platforms, a component that plays a pivotal role is the Memory Controller (MC). It governs the access to the main memory and orchestrates the data transfers between the processing units and the memory devices. Being the main memory a shared resource, real-time tasks running concurrently on different cores may suffer from interference due to memory contention, which depends on both the adopted memory devices and the internal scheduling policies employed by the MC. Despite interesting research on time-predictable MCs [5]–[7], many high-performance, COTS platforms that use DRAM technology still come with MCs employing scheduling policies that aim at maximizing throughput and whose details are not publicly available.

While some large industrial players can obtain this information and build a detailed model of the MC internals, many others simply cannot. As such, numerous designers that intend ensuring real-time guarantees on such platforms struggle in bounding memory-contention delays and are often forced to resort to empirical approaches based on examinations of the system behavior.

Nevertheless, two important observations can be made to tackle this problem. First, although memory contention scenarios are certainly many, they are still limited in number. As such, with principled stimuli and a reasonable amount of time, contention scenarios could be explored to infer the system behavior in detail. Second, although the MC internals are not publicly documented, many fundamental principles of MCs and DRAM behavior are instead known (e.g., per bank queuing, precedence to row hits, fair access to memory banks, technology-related timing constraints, etc.). Thus, this knowledge can be leveraged to both drive the exploration of contention scenarios and design MC models to be refined and eventually trained with experimental data.

Contribution. Motivated by these observations, this paper presents *FrATM^2: a **Fr**amework to **A**utomatically learn the **T**iming **M**odels of the **M**emory subsystem*. Starting from a set of templates, the framework is capable of automatically generating micro-benchmark applications to stimulate memory contention scenarios on a target platform, and then filtering and aggregating the resulting memory access latencies profiled by the micro-benchmarks. The results can be used to both refine the information retrieved by the framework, with the purpose of generating and executing other micro-benchmarks, as well as, eventually, train MC models. Two types of MC models are considered: a coarse-grained model based on either constrained linear regression or a convex hull, and a fine-grained model based on mathematical optimization. Two models were selected to provide alternatives in exploring the trade offs between analytical pessimism and safety.

FrATM^2 is finally evaluated on a state-of-the-art embedded multi-core platform (AMD/Xilinx Ultrascale+) equipped with a DDR4 memory, demonstrating its effectiveness in training the MC models after collecting gigabytes of raw experimental data by testing tents of thousands of contention scenarios

Paper structure. The paper is organized as follows. Section II reports background information. Section III discusses the system model. Section IV presents a high-level overview of FrATM^2, while Section V describes the framework internals and workflow. Section VI presents our experimental evaluation. Section VII discusses the related work. Finally, Section VIII concludes the paper.

II. Essential Background

To make the paper self-consistent, this section briefly recalls background concepts on Dynamic Random-Access Memory (DRAM) memories and ARMv8-A barrier instructions.

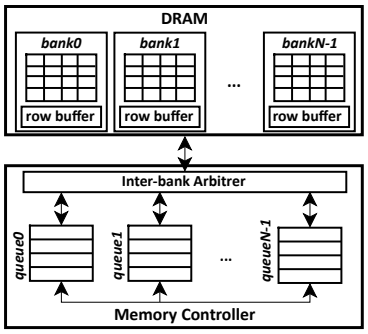

Fig. 1: DRAM and MC interconnection and structure

DRAMs. Figure 1 shows a high-level block diagram of a DRAM and a corresponding Memory Controller (MC). The DRAM subsystem is a collection of memory chips organized in *ranks*, which are in turn organized into multiple *banks*. Each bank is a matrix of a number of *rows* and *columns*, in which each memory cell holds several bytes.

The access to the memory locations is orchestrated by the MC, which receives memory requests from processing elements and issues commands to the DRAM. The MC is connected to the DRAM through two distinct buses: the command and data buses. It is capable of handling memory requests targeting different banks in parallel. Memory addresses, defined in the next section, uniquely identify memory locations. The MC carries out the mapping between them.

Each bank is provided with a temporary buffer called *row buffer*, which acts like a cache by holding the content of the last-accessed row. Memory requests that access the same bank can result in two different scenarios: *row hit* or *row conflict*. A row hit occurs when subsequent memory requests directed to the same bank target the same row loaded in the row buffer. This results in a lower latency due to the caching in the row buffer. Conversely, a row conflict occurs when subsequent accesses to the same bank target different rows. This implies that the content of the row buffer must be copied back to memory before being replaced with the one of a different row. In row hits, columns are accessed by issuing the `CAS` (Column Access Strobe) command. In the other case (row conflict), the access to a column follows the in-order issue of `PRE` (PREcharge), `RAS` (Row Active Strobe), and `CAS` DRAM commands. In more detail, the `PRE` command copies back to the memory what is currently stored in the row buffer; the `RAS` command loads a row from the memory into the row buffer; while eventually the `CAS` command accesses a column by means of read or write request within the row buffer.

To issue these commands, the MC must respect the timing constraints imposed by the JEDEC [8] standard for DRAM memories. These values are also stored in an EEPROM memory chip, available within the DRAM memory, alongside other information about the chip itself, such as the serial number, the manufacturer code, or the revision code. The Serial Presence Detect (SPD) is a standardized way to access this information through the I^2C protocol. This also serves for DRAM automatic configuration.

ARMv8-A barriers. The instruction set of the ARMv8-A processor architecture includes different barrier instructions that enforce memory operation or instruction ordering before the barrier completes. While a detailed description of the ARMv8-A barriers cannot fit in this work, we briefly recall two barrier instructions of our interests: the Data Synchronization Barrier (`dsb`) and the Instruction Synchronization Barrier (`isb`). The `dsb` ensures the completion of all the memory operations before the barrier instruction completes. The `isb` guarantees that all the instructions that follow the barrier are fetched only once the barrier instruction has completed.

III. System model

The main symbols used in this work are summarized in Table I. Throughout the rest of the paper, we consider a system that comprises a set of N_P processing cores (also referred to as CPUs in the following) defined as $\mathcal{C} = \{c_0, c_2, ..., c_{N_P-1}\}$. All the cores share a DRAM memory managed by an MC. A single-rank DRAM is considered for simplicity. The cores can directly issue memory requests, through either loads or stores instructions, directed to specific memory addresses. A memory address A is composed of four different groups of bits that respectively identify the bank, row, column, and offset of a unique memory location. Please note that, due to manufacturer-dependent memory hardware design, the physical address could include some bits that must always be set to a specific value, i.e., 0 or 1. This happens in the context of a memory chip with a non-contiguous address space. Thus, these bits can be safely ignored and, consequently, they are not considered in our definition of the address A. We assume that the bits of each group are contiguous, i.e., the bits belonging to the same group cannot be interleaved with bits of any other group. This assumption is later validated in our experiments.

We denote by bg, rg, cg, and og the groups of bits that identify the banks, rows, columns, and offsets, respectively. The number of bits of each group is respectively defined as N_{bg}, N_{rg}, N_{cg}, and N_{og}.

The DRAM memory comprises $N_B = 2^{N_{\text{bg}}}$ banks defined as $\mathcal{B} = \{B_0, \ldots, B_{N_B-1}\}$. Each bank is a matrix of $N_R = 2^{N_{\text{rg}}}$ rows defined as $\mathcal{R} = \{R_0, \ldots, R_{N_R-1}\}$ and $N_C = 2^{N_{\text{cg}}}$ columns. Each cell of the matrix contains $N_O = 2^{N_{\text{og}}}$ bytes.

Memory controller model. In this paper, we rely on the modeling assumptions for MC adopted in previous work [9], [10], which are also briefly recalled here. They derive from both academic research on MC and the experimental analysis of industrial-grade designs. Given that the internal details of commercial MC are not publicly available, relying on modeling assumptions from sound previous work represents, in our considered opinion, the most pragmatic choice to formally reason about the MC behavior. The MC handles read requests using a queue for each bank. An inter-bank scheduler decides which request to forward to the DRAM memory from the ones available in the per-bank queues. Each request comprises a series of commands, i.e., `PRE`, `RAS`, and `CAS`, described in Section II. A new request is scheduled only once the previous one is completed, if any. The request completes when all its commands have been issued and the corresponding JEDEC timing constraints are met. Conversely,

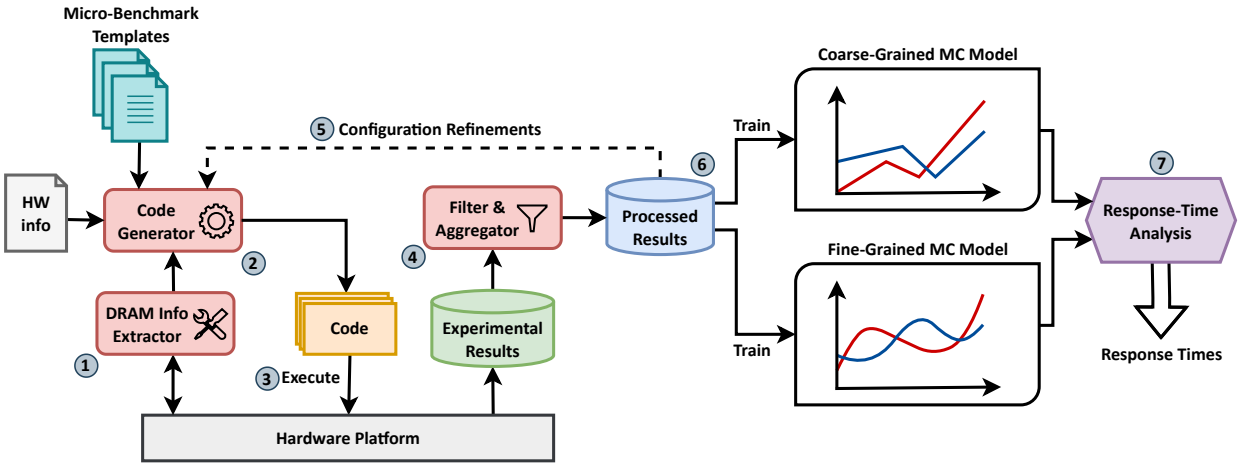

Fig. 2: Block diagram of the proposed framework.

write requests are separately enqueued and served in batches, a technique known as write batching. This prioritizes reads over writes to improve throughput while still avoiding starvation.

The MC behavior is defined by a set of rules classified into three categories: **(i)** intra-queue arbitration, **(ii)** inter-queue arbitration, and **(iii)** write handling. Intra-queue arbitration rules order requests based on the hierarchical policy First-Ready First-Come-First-Served (FR-FCFS). The requests in each queue are arranged in FCFS fashion, but requests resulting in row hits are prioritized over those resulting in row conflicts. The latter is the FR rule, which applies ahead of the FCFS rule for at most N_{thr} requests consecutively. Inter-queue arbitration rules follow the Round-Robin scheduling algorithm, serving only one request per bank at each turn.

Finally, write requests are queued in a dedicated *write buffer*. Write handling rules dictate that, once the number of enqueued requests reaches a threshold W_{thr} (Watermark Threshold), the MC serves at least N_{wb} of them in a single batch (more writes can be included in the batch if no read request is waiting to be served).

For the sole purpose of modeling the MC, like in previous work [9], [10], we assume that **(i)** Q_{write} denotes the known write buffer size, **(ii)** when the watermark threshold (W_{thr}) is exceeded, there are always enough write requests (N_{wb}) to form a batch (i.e., $W_{\text{thr}} \geq N_{\text{wb}}$), **(iii)** the total number of writes drop below the watermark threshold after issuing a batch (i.e., $Q_{\text{write}} - N_{\text{wb}} < W_{\text{thr}}$), and **(iv)** the write buffer is large enough that it can never be full.

IV. Overview of FrATM^2

This section presents a Framework to Automatically learn the Timing Models of the Memory subsystems (FrATM^2), which works by means of automated profiling of a target hardware platform. The resulting models can eventually be used to enable response-time analysis to evaluate the timing properties of, for instance, real-time tasks.

A top-down presentation approach is used, first introducing the framework at a high level in this section and later detail the framework internals in the next section. Figure 2 depicts a block diagram of the framework, also illustrating the various steps it employs (labeled with circled numbers). To train the MC models, it is necessary to properly investigate the DRAM subsystem, gather its settings, and discover its timing behavior by exploring contention scenarios.

The first step ① consists in executing the *DRAM Info Extractor* module, which reads the DRAM's SPD EEPROM to gather its internal configuration parameters (e.g., N_B, N_R, etc., introduced in Sec. III).

This information is required for step ②, in which a *Code Generator* is employed to produce a collection of micro-benchmark applications to be executed on the target hardware platform. The Code Generator leverages a set of templates to generate the C code of the micro-benchmarks, which are written using the Jinja2 [11] template engine, as well as other static information that can be extracted from the datasheet of the target DRAM chip(s). The templates encode a series of configurable memory contention scenarios that, based on previous work and experience, are known to exhibit worst-case timing performance. As such, the templates are meant to be written by experts that are confident with the adopted MC models. To avoid unwanted interference and secure effective time measurements, the micro-benchmarks are executed on bare-metal hardware. The framework provides a minimal execution environment to run micro-benchmarks that includes the power on and setup of **(i)** all processing cores, **(ii)** a serial device, **(iii)** the Generic Interrupt Controller (GIC), and **(iv)** the Memory Management Unit (MMU). The address translation carried out by the MMU goes through a series of translation tables. These tables are initialized to map every virtual address to an identical physical address, i.e., a flat mapping. The micro-benchmark are executed with the data caches disabled to directly stimulate the memory controller.

In step ③, a selection of the micro-benchmarks are executed on the target platform producing a large amount of experimental raw measurements including memory access latencies and the number and type of memory operations. Details on these measurements are provided in the next section.

In step ④, the experimental data is processed by the *Filter & Aggregator* module to produce aggregated statistics. Steps ②, ③, and ④ may be executed multiple times. In fact, a first

execution is often required to infer some configuration options that cannot be retrieved from datasheets and the DRAM Info Extractor module. Typically, they include the number of bits in the memory address that resolve the banks, rows, and columns (bg, rg, etc., introduced in Sec. III).

For this reason, step ⑤ is used to leverage the aggregated statistics produced in the previous step to configure other micro-benchmarks, which can then be activated after a first execution of the preceding steps. This step is also in charge of refining the configuration of the micro-benchmarks, e.g., whenever some expected contention scenario does not manifest during their execution.

In step ⑥, the aggregated statistics can eventually be used to train MC models. In this work, two different MC models are considered: a fine-grained model based on the findings of [9], and a coarse-grained model based on constrained regression and a convex hull (detailed in the next section). The former is based on mathematical optimization and, although being more accurate than the other model, it requires considerable time for both being trained and used. The latter can instead be trained in a matter of seconds and allows computing memory interference bounds very quickly, hence being more suitable for large-scale design exploration campaigns.

Finally, in step ⑦, the MC models can be used to enable response-time analysis, as they can be queried to provide memory-contention-related interference bounds within a time interval of interest.

V. Workflow and Framework Internals

This section presents the internals of FrATM^2 by considering a reference workflow in which the addressing mapping of the target MC is unknown. The considered workflow is composed of three phases:

- **Phase A**: Identification of the address mapping of the target MC (execution of steps ①, ②, ③, and ④).
- **Phase B**: Extraction of memory-contention delays (execution of steps ⑤, ②, ③, and ④).
- **Phase C**: Training of MC models and analysis (execution of steps ⑥ and ⑦).

The three phases are individually detailed in the next subsections.

A. Phase A

In step ① the DRAM Info Extractor reads through the I^{2}C protocol the SPD DRAM EEPROM (see Section II), which behaves as a subordinate device in the protocol. The memory is accessed using an address that comprises a fixed part, which includes the Device Type Identifier Code (DTIC) [12], and a variable part, which includes the serial address selections. The precise address is usually reported in the technical reference manual of the target platform. For instance, this is the case for the AMD/Xilinx Ultrascale+ SoC. The EEPROM contains the values of parameters N_{bg}, N_{rg}, N_{cg}, and N_{og} in a standard format, which serve as inputs for the Code Generator to instantiate a first set of micro-benchmarks.

Identification of the address mapping. In this phase, a set of micro-benchmarks is executed to identify the structure of memory addresses in terms of groups of bits. We denote by π an ordered sequence of the groups of bits bg, rg, cg, and og (see Section III), e.g., the sequence $\langle bg, rg, cg, og\rangle$ specifies that, starting from the most significant bit of the address, the first N_{bg} bits address banks while the last N_{og} ones address the offset. An example configuration of the address mapping from a sequence of bit groups is illustrated in Figure 3. The set Π contains all the possible permutations of such sequences. Only one permutation $\hat{\pi} \in \Pi$ has the groups in the position that reflects the actual mapping implemented by the MC.

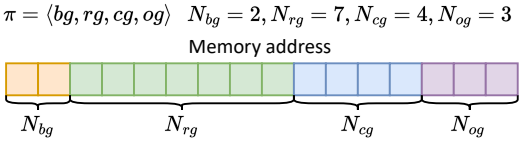

Fig. 3: Example address mapping for a sequence of bit groups.

Since each group can have a different number of bits, given a sequence π the actual position of a bit group in the memory address changes depending on the preceding groups in π. For example, whenever $N_{\text{rg}} \neq N_{\text{cg}}$ holds, the sequences $\pi_1 = \langle rg, bg, cg, og\rangle$ and $\pi_2 = \langle cg, bg, rg, og\rangle$ correspond to different positions of the bits of bg in the memory address, although bg occupies the second position in both π_1 and π_2. For this reason, in principle we are required to test all possible permutations to discover the actual position of a bit group in the memory address.

The micro-benchmarks presented next work by selecting a core $c_0 \in \mathcal{C}$ to suffer interference, while using the other cores $c_z \in \mathcal{C} \setminus \{c_0\}$ to generate interference. Note that this choice is without loss of generality. In fact, since we consider systems with homogeneous cores provided with fair access to the memory subsystem, the results presented in the reminder of the paper also apply whenever the interfered core is different from c_0. For each sequence in Π, c_0 measures the memory access latency while the interfering cores are either idle or generate intensive memory interference. The results allow concluding whether the tested sequence matches the address mapping implemented by the MC based on hypothesis of the expected timing behavior.

Before proceeding, it is important to note that the position of cg and og is not particularly relevant to our purpose, as the memory access latency does not change if two subsequent memory requests target the same row (row-hit) but vary the values of cg, og, or both. Therefore, their position is not required for training the MC models. For this reason, in the following we present two sets of micro-benchmarks for determining the positions of bg and rg only, which are instead required for configuring other micro-benchmarks to be executed in Phase B.

To help the presentation of the algorithms presented next, we define the function `addr`(π, B_i, R_j) that, given a permutation $\pi \in \Pi$, a bank $B_i \in \mathcal{B}$, and a row $R_j \in \mathcal{R}$, returns a memory address $A^{\pi}_{i,j}$ in which the bits in groups bg and rg are set to the bank and row number of B_i and R_j, respectively, while the other bits are set to zero. Furthermore, we define the

function `stride`(π) that, given a permutation $\pi \in \Pi$, returns a memory address s^π where all bits are set to zero except for the least significant bit in rg.

1) Identifying bank and row group bits (bg and rg): A first set of micro-benchmarks is presented to jointly determining the positions of bg and rg bit groups in the memory address. The set of permutations in Π identified by these micro-benchmarks is later handled by other micro-benchmarks, presented in the next subsection, to either refine the selection or double-check the position of the rg bit group.

Two fundamental principles are explored by the micro-benchmarks presented here. First, as memory transactions directed to the same bank are forced to be served *sequentially*, while those directed to different banks can be served *in parallel*, memory-contention delays are expected to be maximized for transactions directed to the same bank issued by different cores. Second, by changing row at each memory access, we can aim at *minimizing the number of row hits*, which in turn aims at both maximizing the delay introduced by contenting requests (due to the issuing of the three commands `PRE`, `RAS`, and `CAS`, see Sec. II) and canceling the effect of the First-Ready rule in intra-bank scheduling employed by the MC.

Building on these principles, the Code Generator (step ②) produces two micro-benchmarks based on the pseudo-code listed in Algorithms 1 and 2. The latter is meant to measure the memory access latency from core c_0 when accessing each bank $B_i \in \mathcal{B}$ (line 12), both in isolation (term Δ_i^{alone} at line 14) and when the other cores access bank $B_j \in \mathcal{B}$ (term $\Delta_{i,j}^{\text{interf}}$ at line 19). All combinations of banks B_i and B_j are tested (note the nested for loop at line 16). Being the banks small in number, due to physical constraints of DRAM chips, and the number of permutations of bit groups limited, the algorithm does not incur scalability issues.

To explore the possible address mapping, the latency measurements are collected for each candidate permutation $\pi \in \Pi$ (line 10), issuing a sufficiently-large number N_Q of read requests to secure the statistical validity of the results.

The LATENCY function (line 2) measures the time elapsed for issuing N_Q memory requests (line 9). Algorithm 1 reports the pseudo-code of the micro-benchmark that generates continuous memory interference (function MAKE_INTERFERENCE) by repeatedly issuing memory reads to addresses that, based on the tested permutation π, are supposed to belong to different rows. This is accomplished by summing the stride s^π value multiple times modulo the maximum number of rows N_R (efficiently implemented with a bit-wise AND operator at line 6). Note that also the LATENCY function repeatedly accesses different rows. This is done to minimize row hits, as introduced at the beginning of the subsection. The **rem_start** command in Alg. 2 (line 18) denotes the activation of a function on the other cores, keeping it running until a **rem_stop** command is executed. The latency results for each permutation π are eventually stored (**store** keyword in the pseudo-code).

To find out which permutations $\pi \in \Pi$ have the bg and rg bits in the position that reflects the actual address mapping of the target MC, after the execution of the micro-benchmarks (step ③), the Filter and Aggregator (step ④) prunes out the permutations that do not satisfy the following invariants.

Algorithm 1 Interference generator

```
1: ▷ Interfering cores, i.e., c_z ∈ C \ {c_0}, generate continuous memory interference. ◁
2: function MAKE_INTERFERENCE(A*, s^π)
3:     A^end ← A* + s^π × (N_R − 1)
4:     loop
5:         issue a read request to address A*
6:         A* ← (A* + s^π) & A^end
```

Algorithm 2 Latency profiler (executed by c_0)

```
1: inputs N_Q
2: function LATENCY(A*, N_Q, s^π)
3:     A^end ← A* + s^π * (N_R − 1)
4:     start ← get_time()
5:     for counter = {1, ..., N_Q} do
6:         issue a read request to address A*
7:         A* ← (A* + s^π) & A^end
8:     dsb()                                   ▷ See Section II
9:     return get_time() − start
10: for all π ∈ Π do
11:     s^π = stride(π)
12:     for all B_i ∈ B do
13:         A^π_{i,0} = addr(π, B_i, R_0)
14:         Δ^alone_i ← LATENCY(A^π_{i,0}, N_Q, s^π)
15:         store (π, Δ^alone_i)
16:         for all B_j ∈ B do
17:             A^π_{j,0} = addr(π, B_j, R_0)
18:             rem_start MAKE_INTERFERENCE(A^π_{j,0}, s^π)
19:             Δ^interf_{i,j} ← LATENCY(A^π_{i,0}, N_Q, s^π)
20:             rem_stop
21:             store (π, Δ^interf_{i,j})
```

Invariant 1.

$$\forall B_x \in \mathcal{B}, \quad \Delta_{x,x}^{\text{interf}} > \max_{\substack{B_i \in \mathcal{B}, B_j \in \mathcal{B} \\ B_i \neq B_j}} \left\{ \Delta_{i,j}^{\text{interf}} \right\}. \tag{1}$$

The invariant specifies that the latency when the interfered core and the interfering cores access the same bank is larger than the latency when the interfered and interfering cores access different banks, for any pair of different banks. If a permutation has the bg and rg bits in the right position, the invariant needs to hold because interference is maximized when transactions are directed to the same bank due to serialization, and the difference with respect to the case in which different banks are accessed is amplified because the number of row hits is minimized.

Invariant 2.

$$\forall B_i \in \mathcal{B}, \forall B_j \in \mathcal{B} \mid B_i \neq B_j, \quad \Delta_{i,i}^{\text{interf}} \simeq \Delta_{j,j}^{\text{interf}}. \tag{2}$$

The invariant specifies that the latency when the interfered core and the interfering cores access the same bank is always almost the same, irrespective of the targeted bank. It needs to hold because the amount of intra-bank interference must be almost the same for all banks, as they are expected to be fairly managed by the MC (due to round-robin inter-bank scheduling).

Invariant 3.

$$\begin{aligned} &\forall B_i \in \mathcal{B}, \forall B_j \in \mathcal{B} \mid B_i \neq B_j, \\ &\forall B_x \in \mathcal{B}, \forall B_k \in \mathcal{B} \mid B_x \neq B_k, \\ &\Delta^{\text{interf}}_{i,j} \simeq \Delta^{\text{interf}}_{x,k}. \end{aligned} \tag{3}$$

The invariant is analogous to Inviariant 2 but for the case in which the interfered core and the interfering cores access different banks. Again, as banks are expected to be fairly managed by the MC, the amount of inter-bank interference must be almost the same, irrespective of the targeted banks.

Invariant 4.

$$\forall B_i \in \mathcal{B}, \forall B_j \in \mathcal{B}, \forall B_k \in \mathcal{B}, \quad \Delta^{\text{alone}}_i < \Delta^{\text{interf}}_{k,j}. \tag{4}$$

The invariant specifies that the latency in accessing any bank in isolation (i.e., without interference) is lower than all cases in which both intra-bank ($B_j = B_k$) and inter-bank ($B_j \neq B_k$) interference is stimulated for any pair of banks.

Outcome. This process returns a subset of permutations $\overline{\Pi} \subset \Pi$ that are filtered by the four invariants above. It allows complementing the information retrieved by the DRAM Info Extractor (step ⑤) to enable other micro-benchmarks. If any two sequences in $\overline{\Pi}$ do not have all the bits of the bg and rg groups in the same position of the memory address, then it is not possible to conclude the identification of the address mapping. In these cases, the micro-benchmark presented in the next subsection serves the purpose of resolving the address mapping by focusing on the rg bit group. Otherwise, the next micro-benchmark can be executed to double-check the address mapping retrieved with the above process.

2) Identifying row group bits (rg): The principle leveraged here is that row hits lead to shorter memory access latency than row conflicts. Therefore, to identify the position of rg bits, for all the remaining permutations in $\overline{\Pi}$ and all pairs of rows, the memory access latency is measured from core c_0, both in isolation and under contention generated by the other cores.

To this end, the Code Generator (step ②) produces a micro-benchmark based on the pseudo-code listed in Algorithm 3. The MAKE_INTERFERENCE and LATENCY functions are the same defined in Algorithms 1 and 2, respectively. Note that the stride parameter s^π is kept to zero (line 3) to avoid changing row in the memory accesses performed by these two functions. To improve the scalability of the algorithm, we introduce the set $\widehat{\mathcal{R}}$ to optionally enable the exploration of a subset of all possible rows (defined later in Sec. VI) only. All pairs of rows are explored with nested loops (lines 4 and 8), collecting both the results in isolation (term δ^{alone}_i at line 6) and under contention (term $\delta^{\text{interf}}_{i,j}$ at line 11) accessing memory addresses that are supposed to belong to different R_i and R_j of the same bank B_0 (any bank could have been chosen).

To find out which permutations $\pi \in \overline{\Pi}$ have the rg bits in the right position, after the execution of the micro-benchmark (step ③), the Filter and Aggregator (step ④) verifies the following invariants.

Invariant 5.

$$\forall R_x \in \mathcal{R}, \ \delta^{\text{interf}}_{x,x} < \min_{\substack{R_i \in \mathcal{R}, R_j \in \mathcal{R} \\ R_i \neq R_j}} \left\{ \delta^{\text{interf}}_{i,j} \right\}. \tag{5}$$

Algorithm 3 Latency profiler (executed by c_0)

```
1:  inputs N_Q
2:  for all π ∈ Π̄ do
3:      s^π = 0
4:      for all R_i ∈ R ∩ R̂ do
5:          A^π_{0,i} = addr(π, B_0, R_i)
6:          δ^alone_i ← LATENCY(A^π_{0,i}, N_Q, s^π)
7:          store (π, δ^alone_i)
8:          for all R_j ∈ R ∩ R̂ do
9:              A^π_{0,j} = addr(π, B_0, R_j)
10:             rem_start MAKE_INTERFERENCE(A^π_{0,j}, s^π)
11:             δ^interf_{i,j} ← LATENCY(A^π_{0,i}, N_Q, s^π)
12:             rem_stop
13:             store (π, δ^interf_{i,j})
```

The invariant specifies that the latency when the interfered core and the interfering cores access the same row of the same bank is smaller than the latency when the interfered and interfering cores access different rows. The invariant needs to hold because accesses to a row already loaded in the row buffer (row hit) is done by means of `CAS` commands only, while accesses to a different row (row conflict) requires the in-order execution of `PRE`, `RAS`, and `CAS` commands, hence leading to a larger latency.

Invariant 6.

$$\forall R_i \in \mathcal{R}, \forall R_j \in \mathcal{R} \mid R_i \neq R_j, \ \delta^{\text{interf}}_{i,i} \simeq \delta^{\text{interf}}_{j,j}. \tag{6}$$

The invariant specifies that the latency when the interfered core and the interfering cores access the same row of the same bank is always almost the same, irrespective of the targeted row. It needs to hold because the time to access to the row buffer does not depend on the row loaded into it.

Invariant 7.

$$\begin{aligned} &\forall R_i \in \mathcal{R}, \forall R_j \in \mathcal{R} \mid R_i \neq R_j, \\ &\forall R_x \in \mathcal{R}, \forall R_k \in \mathcal{R} \mid R_x \neq R_k, \\ &\delta^{\text{interf}}_{i,j} \simeq \delta^{\text{interf}}_{x,k}. \end{aligned} \tag{7}$$

The invariant is analogous to Invariant 6, but considering accesses to different rows. It needs to hold because row conflicts require the in-order execution of `PRE`, `RAS`, and `CAS` commands, which have the same timing constraints independently of the targeted row.

Invariant 8.

$$\forall R_i \in \mathcal{R}, \forall R_j \in \mathcal{R}, \ \delta^{\text{alone}}_i \simeq \delta^{\text{interf}}_{j,j}. \tag{8}$$

The invariant specifies that the latency either when the interfered core runs in isolation or when the interfered core and the interfering cores access the same row is always almost the same, irrespective of the targeted row. It needs to hold because in both cases the memory access is performed by means of a `CAS` command only (row hits) and the JEDEC standard mandates a very low timing constraint between `CAS` commands, in the order of a few nanoseconds. For instance, on a typical DDR4, the constraint is about one ninth of the cumulative constraint for row misses (Invariant 5). As such,

interfering row hits are expected to introduce delays that are not observable with the micro-benchmarks.

Outcome. Inviariants 5-8 allow identifying a refined subset of permutations $\widehat{\Pi} \subset \overline{\Pi}$ that is expected to include only sequences with bits bg and rg in the same position. This information can hence be used to refine the configuration of the Code Generator (step ⑤) to enable other micro-benchmarks, which are presented in the next subsection. In the unfortunate case in which $\widehat{\Pi}$ still includes sequences with conflicting positions of the bg and rg bits, the identification of the addressing mapping fails and the whole process implemented by the framework is aborted.

B. Phase B

This phase is concerned with the execution of micro-benchmarks to compute the interference induced by memory contention starting from the cumulative memory access latency experienced by a core $c_0 \in \mathcal{C}$. Again, all other cores $c_z \in \mathcal{C} \setminus \{c_0\}$ are used to generate memory interference.

Differently from Phase A, the micro-benchmarks presented next take multiple measurements by varying the type of memory requests. The type can be either **(i)** a memory read (P_1), **(ii)** a memory write (P_2), or **(iii)** a read or a write randomly selected with uniform distribution (P_3). The collection of all types is denoted by $\mathcal{P} = \{P_1, P_2, P_3\}$.

To both explore heterogeneous contention scenarios and investigate how memory interference varies as the number of memory requests issued by the interfered core increases, multiple *experimental campaigns* are performed by executing the same micro-benchmark code multiple times with different numbers of memory requests N_Q. We denote by $\mathcal{Q}$ the ordered multiset of the numbers of requests tested in each campaign, where $\mathcal{Q}[i]$ is the value for the i-th campaign.

Measurements are obtained by issuing requests to a wide range of memory addresses. To maximize coverage with the tested contention scenario, requests are issued to pseudo-randomized addresses, which are efficiently generated with the Multiplicative Linear Congruential Generator (MLCG) [13] algorithm. Function `rand()` in the pseudo-code presented next implements the MLCG algorithm. The pseudo-random generator generates a number starting from another number, which is passed as a parameter to the `rand()` function. It hence allows generating a chain of numbers where the last generated number is used to generate the next one. Each generated number is eventually used to generate the address of a memory request. Therefore, in the i-th campaign, $\mathcal{Q}[i]$ numbers are generated. The numbers of the chain generated for core c_z during the i-th campaign are denoted by $\omega^z_{i,1}, \omega^z_{i,2}, \ldots, \omega^z_{i,j}, \ldots$, where $\omega^z_{i,j+1} = \texttt{rand}(\omega^z_{i,j}), \forall j \geq 1$. The chain needs to start with a number $\omega^z_{i,0}$ provided by the user that serves as a *seed* and allows ensuring reproducibility of the results across different runs, i.e., $\omega^z_{i,1} = \texttt{rand}(\omega^z_{i,0})$. One seed per campaign is randomly generated offline by the Code Generator (step ②) and provided in input to the micro-benchmarks.

For each campaign with index i, the interfered core c_0 issues $\mathcal{Q}[i]$ requests, hence the corresponding chain of pseudo-random numbers ends with $\omega^0_{i,\mathcal{Q}[i]}$. Conversely, the interfering cores are required to generate continuous memory accesses with an unbounded number of memory requests. Hence, the corresponding chains are unbounded as well.

Function `to_addr`$(\omega^z_{i,j})$ is used to obtain a valid memory address from a number in a chain by applying a bit mask. The micro-benchmarks stimulate the memory subsystem by issuing memory requests interleaved with some CPU computation, to mimic the behavior of realistic tasks. To this end, we use the function `delay()` to introduce a random number of NOP instructions after each request. Finally, to extract the bank number from a memory address, we define the function `bank_of`(A) that returns the bits in bg of an address A. This operation can be performed because the position of bg is known from Phase A.

Algorithms. The Code Generator (step ②) produces two micro-benchmarks based on the pseudo-code listed in Algorithms 4 and 5. The latter is meant to measure the Cumulative Memory Access Time (CMAT) from core c_0 to each address obtained from pseudo-random numbers generated from the seed $\omega^0_{i,0}$ (line 16). Memory is accessed by issuing memory requests of type $P_h \in \mathcal{P}$ (line 17), both in isolation (term $t^{\text{alone}}_{i,h}$ at line 18) and when the other cores issue memory requests of type $P_l \in \mathcal{P}$ (term $t^{\text{interf}}_{i,h,l}$ at line 23). All combinations of types P_h and P_l are tested (note the nested loop at line 20). Being $\mathcal{P}$ fixed, the scalability of this algorithm depends on $|\mathcal{Q}|$ only.

The CMAT function (line 5) measures the time elapsed for issuing P_h memory requests to each address of a given sequence (line 7). Algorithm 4 reports the pseudo-code of the micro-benchmark that generates continuous memory interference (function MAKE_INTERFERENCE). At each iteration, it calculates the new seed (line 5) used to determine the address (line 6) to which it issues the P_l memory request (line 8).

For each pair of request types, the micro-benchmarks count the number of memory reads and writes issued by each core $c_z \in \mathcal{C}$ to each bank $B_k \in \mathcal{B}$, which are stored in variables $\eta^r_{i,z,k}$ and $\eta^w_{i,z,k}$, respectively (Algorithm 4, line 9 and Algorithm 12, line 12). The **inc** keyword in the pseudo-code denotes the increment of a variable. As for the algorithms presented in Phase A, the **rem_start** and **rem_stop** commands activate and deactivate a function on the other cores, respectively. The CMAT results are eventually stored in a tuple alongside variables $\eta^r_{i,z,k}$ and $\eta^w_{i,z,k}$ (**store** keyword in the pseudo-code). Note that the latter two variables are recomputed for each tested pair of request types (P_h, P_l) and are hence individually correlated with every parameter $t^{\text{interf}}_{i,h,l}$ (note indeed the RESET function in Algorithm 5).

To secure the statistical validity of the measurements, all experimental campaigns are repeated N_T times. It is important to note that, each of these N_T times, will issue a predictable sequence of memory accesses (both in terms of addresses and types) determined by the seeds used by the pseudo-random generator. This enables meaningful aggregation and comparison of the results obtained across different repetitions.

For each campaign with index i, the Filter and Aggregator (step ④) calculates the memory interference estimate $I_{i,h,l}$ over the N_T samples for all the combinations of access types $P_h, P_l \in \mathcal{P}$. If $t^{\text{alone}}_{i,h}[v]$ and $t^{\text{interf}}_{i,h,l}[v]$ are the CMAT values in isolation and with interference, respectively, measured at

Algorithm 4 Interference generator

1: ▷ *Interfering cores, i.e., $c_z \in \mathcal{C} \setminus \{c_0\}$, generate continuous memory interference.* ◁
2: **inputs** $\omega^z_{i,0}, \forall i, z > 0$
3: **function** MAKE_INTERFERENCE(i, P_l)
4: **for** $j = 1, \ldots$ **do** ▷ *infinite loop*
5: $\omega^z_{i,j} = \texttt{rand}(\omega^z_{i,j-1})$
6: $A = \texttt{to\_addr}(\omega^z_{i,j})$
7: $k = \texttt{bank\_of}(A)$
8: issue a P_l memory request to address A
9: **inc** $\eta^r_{i,z,k}$ or $\eta^w_{i,z,k}$ according to P_l

Algorithm 5 CMAT profiler (executed by c_0)

1: **inputs** $\mathcal{Q}$, $\omega^z_{i,0}, \forall i, z$
2: **function** RESET(i)
3: **for all** $c_z \in \mathcal{C}$, $B_k \in \mathcal{B}$ **do**
4: $\eta^r_{i,z,k}, \eta^w_{i,z,k} \leftarrow 0, 0$
5: **function** CMAT($\omega^0_{i,0}, N_Q, P_h$)
6: $start \leftarrow \texttt{get\_time()}$
7: **for all** $j = 1, 2, \ldots, N_Q$ **do**
8: $\omega^0_{i,j} = \texttt{rand}(\omega^0_{i,j-1})$
9: $A = \texttt{to\_addr}(\omega^0_{i,j})$
10: $k = \texttt{bank\_of}(A)$
11: issue a P_h memory request to address A
12: **inc** $\eta^r_{i,0,k}$ or $\eta^w_{i,0,k}$ according to P_h
13: `delay()`
14: `dsb()`
15: **return** $\texttt{get\_time()} - start$
16: **for all** $i = 1, \ldots, |\mathcal{Q}|$ **do** ▷ *For each campaign*
17: **for all** $P_h \in \mathcal{P}$ **do**
18: $t^{\text{alone}}_{i,h} \leftarrow \text{CMAT}(\omega^0_{i,0}, \mathcal{Q}[i], P_h)$
19: **store** $(\omega^0_{i,0}, t^{\text{alone}}_{i,h})$
20: **for all** $P_l \in \mathcal{P}$ **do**
21: RESET(i)
22: **rem_start** MAKE_INTERFERENCE(i, P_l)
23: $t^{\text{interf}}_{i,h,l} \leftarrow \text{CMAT}(\omega^0_{i,0}, \mathcal{Q}[i], P_h)$
24: **rem_stop**
25: **store** $\left(t^{\text{interf}}_{i,h,l}, \bigcup_{\substack{c_z \in \mathcal{C} \\ B_k \in \mathcal{B}}} \{\eta^r_{i,z,k}, \eta^w_{i,z,k}\} \right)$

the v-th repetition of the execution of the campaigns, the interference estimate is computed as

$$I_{i,h,l} = \max_{v=1,\ldots,N_T} \left\{ t^{\text{interf}}_{i,h,l}[v] \right\} - \max_{v=1,\ldots,N_T} \left\{ t^{\text{alone}}_{i,h}[v] \right\}. \quad (9)$$

The rational behind the above equation is to estimate interference, for each campaign (subscript i) and combination of access types (subscripts h and l), as the difference between the longest-observed CMAT when the interfering cores are active and the longest-observed CMAT in isolation.

Outcome. This phase returns a substantial amount of data composed of **(i)** all the interference estimates $I_{i,h,l}$ computed by Eq. (9) and **(ii)** for each $I_{i,h,l}$, all numbers of memory accesses $\eta^r_{i,z,k}$ and $\eta^w_{i,z,k}$ stored by Alg. 5 (line 25) in the same tuple of the parameter $t^{\text{interf}}_{i,h,l}[v]$ that is maximal in Eq. (9). The resulting dataset is therefore consisting of a series of tuples of the form:

$$\left(I_{i,h,l} \Rightarrow \bigcup_{c_z \in \mathcal{C}, B_k \in \mathcal{B}} \{\eta^r_{i,z,k}, \eta^w_{i,z,k}\} \right). \quad (10)$$

This dataset is propagated to Phase C to train two MC models (step ⑥).

C. Phase C

Next, we distinguish between the two MC models we considered. Once trained, these models can eventually be used to enable response-time analysis (step ⑦), as for instance suggested by previous work [9], [10].

1) Training the coarse-grained model: The first model aims at quantifying interference based on **(i)** the number of reads and writes issued by the interfered core; and **(ii)** the number of reads and writes issued by the other interfering cores. The vector $\boldsymbol{\eta}$ is introduced to contain these numbers, in the same order as written above, and denote the input parameters of the interference function $\mathcal{I}(\boldsymbol{\eta})$ to be learned from the dataset exported by Phase B.

To train our MC model it is convenient to further aggregate the data coming from the previous phase. First, for each campaign with index i, we aggregate the number of reads and writes issued to the various banks by the interfered core c_0:

$$\eta^r_{i,0} = \sum_{B_k \in \mathcal{B}} \eta^r_{i,1,k}, \qquad \eta^w_{i,0} = \sum_{B_k \in \mathcal{B}} \eta^w_{i,1,k}. \quad (11)$$

Second, we do the same for all the interfering cores:

$$\eta^r_{i,*} = \sum_{c_z \in \mathcal{C} \setminus \{c_0\}} \sum_{B_k \in \mathcal{B}} \eta^r_{i,z,k}, \quad \eta^w_{i,*} = \sum_{c_z \in \mathcal{C} \setminus \{c_0\}} \sum_{B_k \in \mathcal{B}} \eta^w_{i,z,k}. \quad (12)$$

Given the relationship in Equation (10) that binds $I_{i,h,l}$ to parameters $\eta^r_{i,z,k}, \eta^w_{i,z,k}$, it is also possible to bind each value $I_{i,h,l}$ to those in Equations (11) and (12). Therefore, a vector $\boldsymbol{\eta}_{i,h,l} = [\eta^r_{i,0}, \eta^w_{i,0}, \eta^r_{i,*}, \eta^w_{i,*}]$ can also be associated with each interference estimate $I_{i,h,l}$. This allows defining our final dataset for training as a list of tuples of the form

$$(I_{i,h,l} \Rightarrow \boldsymbol{\eta}_{i,h,l}). \quad (13)$$

We are interested in learning an interference function $\mathcal{I}(\boldsymbol{\eta})$ that satisfies $\mathcal{I}(\boldsymbol{\eta}_{i,h,l}) \geq I_{i,h,l}, \forall i, h, l$. To do so, we present two methods: one based on constrained regression and another one based on a convex hull.

Constrained regression. This method aims at learning a linear function representing a hyperplane defined by the following equation:

$$\mathcal{I}(\boldsymbol{\eta}) = W \cdot \boldsymbol{\eta}^{\intercal} + b, \quad (14)$$

where $W \in \mathbb{R}^{4+}$ and $b \in \mathbb{R}^{+}$ represent the slope matrix and the intercept of the hyperplane[1]. These terms can assume only positive values because the interference grows as the

[1]We denote with $\mathbb{R}^{+}$ the set of the positive real numbers, zero included.

TABLE I: Table of main symbols.

Sym.	Description	Sym.	Description
c_z	z-th physical core	bg, rg	Groups of bits that identify the banks and rows, respectively
B_k	k-th memory bank	$\mathcal{C}$, $\mathcal{B}$	Set of cores and banks, respectively
R_j	j-th memory row	$\eta^r_{i,z,k}$, $\eta^w_{i,z,k}$	Reads and writes of c_z on B_k during i-th campaign
A	Memory address	$\eta^r_{i,0}$, $\eta^r_{i,*}$	Reads of interfered and interfering cores during i-th campaign
N_P	Number of cores	$\mathcal{Q}$	Multiset of the numbers of requests per campaign
N_Q	Number of memory requests	$\mathcal{Q}[i]$	Num. of mem. reqs. of i-th campaign
N_T	Micro-benchmark repetitions	$t^{\text{alone}}_{i,h}$	CMAT of i-th campaign in isolation issuing P_h mem. reqs.
$\mathcal{P}$	Set of memory request types	$t^{\text{interf}}_{i,h,l}$	CMAT of i-th campaign issuing P_h mem. reqs. with interfering P_l mem. reqs.
$\omega^z_{i,0}$	Seed of c_z for i-th campaign	$\omega^z_{i,j}$	j-th random number of c_z during the i-th campaign
P_h	h-th memory request type	$\boldsymbol{\eta}_{i,h,l}$	Numbers of reads and writes of the interfered and interfering cores associated to $I_{i,h,l}$
$\mathcal{I}(\boldsymbol{\eta})$	Learned interference function	$I_{i,h,l}$	Interference estimate of i-th campaign with P_h and P_l mem. reqs. issued by the interfered and interfering cores, respectively

components of $\boldsymbol{\eta}$ increase. The constrained regression is calculated to minimize the square distance between the interference estimates measured in Phase B and their projection to the hyperplane defined by Equation (14). Therefore, the training of both W and b consists in minimizing the following cost:

$$\sum_{i=1}^{|\mathcal{Q}|} \sum_{P_h \in \mathcal{P}} \sum_{P_l \in \mathcal{P}} \left[\mathcal{I}(\boldsymbol{\eta}_{i,h,l}) - I_{i,h,l}\right]^2. \tag{15}$$

Furthermore, the interference function $\mathcal{I}(\boldsymbol{\eta})$ must be constrained to be always larger than all the experimental measurements, hence the following constraint needs to be satisfied:

$$\forall i = 1, \ldots, |\mathcal{Q}|, \forall P_h \in \mathcal{P}, \forall P_l \in \mathcal{P},\ \mathcal{I}(\boldsymbol{\eta}_{i,h,l}) \geq I_{i,h,l}. \tag{16}$$

Convex hull. While being simple to understand and easy to train, the model based on constrained regression may be too pessimistic. For this reason, we present a second method based on a convex hull to obtain a piece-wise multidimensional function that provides a tighter bound of the memory interference. The convex hull that contains all the interference estimates in the training dataset (Eq. (13)) can be defined by the minimal set of facets $\mathcal{H} = \{H_1, H_2, \ldots\}$ that guarantees

$$\begin{aligned} &\forall i = 1, \ldots, |\mathcal{Q}|, \forall P_h \in \mathcal{P}, \forall P_l \in \mathcal{P}, \\ &\forall H_x \in \mathcal{H},\ |\ W_x \cdot [\boldsymbol{\eta}_{i,h,l}, I_{i,h,l}]^\intercal + b_x \leq 0, \end{aligned} \tag{17}$$

where $W_x \in \mathbb{R}^5$ and $b_x \in \mathbb{R}$. The Quickhull [14] algorithm can be used to compute the parameters W_x and b_x of the facets, hence training the convex hull from experimental data. The interference function $\mathcal{I}(\boldsymbol{\eta})$ can then be obtained by computing the projection of $\boldsymbol{\eta}$ on the hyperplanes of the upper, non-descending facets of the convex hull.

2) Training the fine-grained model: As an alternative approach, we explored the possibility to train the fine-grained MC model based on mathematical optimization presented in [9] (briefly summarized in Sec. III) with the experimental data produced by the proposed framework. This MC model is designed to theoretically bound the maximum memory interference that can be suffered by the interfered core.

Given the theoretical approach at the core of the MC model presented in [9], it is important to note that the latter copes with rare worst-case scenarios that are extremely hard (if not impossible) to observe experimentally. Furthermore, still to provide conservative and safe results, it limitedly accounts for pipelining in serving memory requests and transforming data. Finally, the model from [9] is tied to a scheduling hierarchy that, while in principle expected to be reasonably similar to the one of the profiled MC, may differ by a string of details that are not publicly documented.

As such, the objective of this training is to find the parameters of this fine-grained MC model so that it can produce results capable of bounding the experimental data, while not pretending to learn the actual internal parameters of the profiled MC. Consequently, the bounds generated by the fine-grained model are inherently more pessimistic, albeit safer, than those generated by the coarse-grained model.

The optimization problem on which this model is based takes as input the number of read requests emitted by all the cores $c_z \in \mathcal{C}$ to each bank $B_k \in \mathcal{B}$ for all the address sequences (i.e., parameters $\eta^r_{i,z,k}$ from Eq. (10)), as well as the number of write requests $\eta^w_{i,z}$ emitted by every core $c_z \in \mathcal{C}$, independently from the target bank, which is calculated using the following equation: $\eta^w_{i,z} = \sum_{B_k \in \mathcal{B}} \eta^w_{i,z,k}$. The constraints of the optimization problem characterize the possible interference scenarios, in accordance with the arbitration rules of the memory controller model introduced in Section 3. As a result, the output of the optimization is the worst-case memory interference that can be suffered in the depicted scenario.

As introduced in Sec. III, the MC model derived from [9] depends on four architectural parameters (i.e., N_{thr}, W_{thr}, N_{wb}, and Q_{write}). Since these parameters are not known a priori, the goal of the training phase is to find a set of values so that the implied worst-case interference correctly upper bounds the experimental data.

For this reason, we developed an iterative algorithm to train such values. Before starting to describe it, we recall that, for consistency, we need $W_{\text{thr}} \geq N_{\text{wb}}$. Also, remember that the lower the value of the parameters, the lower is the result of the optimization.

Training algorithm. Initially, we set $N_{\text{thr}} = W_{\text{thr}} = N_{\text{wb}} = Q_{\text{write}} = 1$. Then, we start iterating over the tuples obtained from Phase B (Eq. (10)), and, using the current values of the parameters, we query the MC model to find the maximum theoretical memory interference with the current configuration. If the theoretical value is larger than the measured value, then the bound is respected, and we move to the next tuple. Otherwise, it means that the interference bound imposed by the model is incorrect with the current set of parameters. Therefore, we increase the values of one of them by 1.

The exact parameter to be incremented is chosen with a round-robin approach (i.e., every time a different parameter is increased). We then repeat the test of the tuple with the new configuration and we proceed as already described.

Once all the experimental data has been processed, we are sure that the MC model, with the set of parameters that was obtained, can correctly produce memory-interference values that bound the entire set of observations.

VI. Experimental evaluation

This section reports the results of the three phases presented in the previous section. The micro-benchmarks were executed on an AMD/Xilinx Ultrascale+ ZCU102 [15], which features a quad-core Arm® Cortex®-A53 processor and 4 GB DRAM Micron MTA4ATF51264HZ [16]. The experimental evaluation requires executing micro-benchmarks for days, collecting gigabytes of data. The highlights are reported in Table II. The execution time of Phase B grows with the increase of $|\mathcal{Q}|$.

TABLE II: Highlights of the experimental evaluation.

Description	Exec. time	Raw data	Aggreg. data
Phase A	about 5 minutes	150 KB	-
Phase B	about 3 days	2.5 GB	-
Phase C-regression	about 2 seconds	-	53 MB
Phase C-qhull	about 2 seconds	-	53 MB
Phase C-fine-grain	about 70 minutes	-	53 MB
Number of campaigns	$\|\mathcal{Q}\| = 19000$		

The DRAM Info Extractor was implemented in C. The Code Generator was implemented with Jinja2 and produces the micro-benchmarks composed of mostly C code, with the exception of the code segments to issue memory requests that are in assembly. Both the DRAM Info Extractor and the micro-benchmarks were compiled using the aarch64-gcc compiler. The Filter and Aggregator was implemented in Python leveraging the NumPy and SciPy libraries. The training of MC models was implemented in Python as well. The models were trained on a Dell OptiPlex 7070 that features a Intel® Core™ i7-9700 and 32 GB RAM.

Next, we present the results of Phase A. Then, we jointly present the results of Phases B and C.

A. Results of Phase A

By means of 2D plots, we report the results produced by Algorithms 2 and 3 to discover the address mapping (position of bits bg and rg). Figure 4 depicts the results for which Invariants 1-4 hold. The x-axis and y-axis report the targeted banks and the memory access latencies in us, respectively, over $N_Q = 1000$ memory requests measured by c_0. One line per bank accessed by the interfering cores $c_z \in \mathcal{C} \setminus \{c_0\}$ is reported. The line labeled with "ALONE" depicts the latency when the interfered core runs in isolation (Alg. 2 line 14 and Alg. 3 line 6). Note that each plot refers to a permutation where the bits of bg, rg, cg, and og are denoted with contiguous sequence of '`b`', '`r`', '`c`', and '`o`', respectively, reported above the plots. The zeros within the permutation are due to a static memory segmentation employed by the target hardware (see Sec. III). As it can be observed from the plots, latency peaks were recorded when the bank accessed by the interfering and interfered cores is the same.

Figure 5 depicts the results for which Invariants 5-8 hold. The DRAM Info Extractor retrieved 16 bits in rg for the target platform, making it hard to plot all combinations. For this reason and to enhance the scalability of Alg. 3, we focused on varying three consecutive bits at a time in rg (still denoted by 'r' in the labels above the plots), keeping the other bits in rg to zero (indicated with a bold '**0**'). This was obtained by generating a set $\widehat{\mathcal{R}}$ for Alg. 3 by **(i)** taking a sliding window of three bits in rg, **(ii)** generating all possible combinations for such bits, leaving the others set to zero, and **(iii)** producing all row numbers by moving the sliding window in all positions in rg. This ends up in showing the results when the interfered and interfering cores access a subset of the available rows. The same exact trends were observed for all other rows not reported in the plots. The y-axis of Fig. 5 reports the measured access latency in us over $N_Q = 1000$ memory requests. The x-axis and the lines report the values taken by the subset of the rg's bits in the addresses used by the interfered and interfering cores. As it can be observed from the plots, latency down-peaks were recorded when the row accessed by the interfering and interfered cores are the same (row hits).

Overall, we can conclude the address mapping of the target MC needs to start with the rg bit group followed by the bg one. This information is sufficient to proceed to Phase B.

B. Results of Phases B and C

Since the learned interference functions are multidimensional, and hence hard to graphically represent, in this section we restrict to the case of read requests only to produce 3D plots. The x-axis of the plots reports $\eta^r_{i,0}$, the y-axis reports $\eta^r_{i,*}$, and the z-axis either the experimental interference estimates collected in Phase B or the interference bounds produced by the MC models after training.

Figure 6 depicts the experimental interference estimates (grey point clouds, computed by Eq. (9)) alongside the bounds of the coarse-grained model. The number of read requests of each campaign varies in the range $\{10, 30, 50, 100, 200, 300, 500, 750, 1000\}$ (x-axis values). The micro-benchmark of Alg. 5 was repeated $N_T = 100$ times. The red plane represents the interference function obtained with constrained regression, while the blue surface represents one obtained with the convex hull. As expected, the red surface is linear while the blue one expresses a tighter bound. We trained the coarse-grained model using the 85% of the aggregated data, while the remaining 15% was used for validation. The latter shows an accuracy of $99,99\%$ and $99,97\%$ for constrained regression and convex hull, respectively. Figure 7 reports the same experimental data alongside the theoretical interference bounds produced by the fine-grained MC model, for the same numbers of read requests recorded in Phase B, achieving 100% accuracy. Comparing this figure with Fig. 6, we can note how the fine-grained MC model is more pessimistic than the coarse-grained one, since it accounts for rare worst-case scenarios that are hard to observe experimentally. As reported in Tab. II, training the coarse-grained model requires much less time than the fine-grained one.

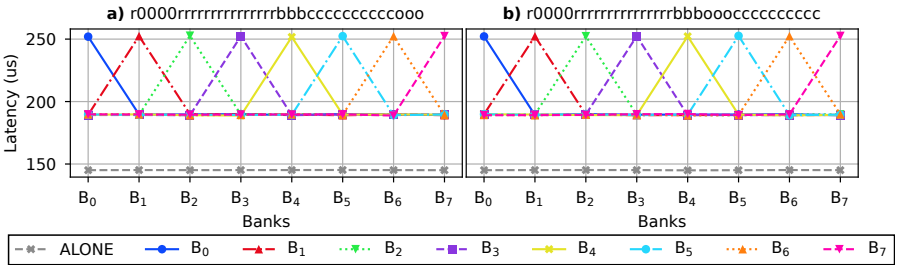

Fig. 4: Results of Phase A: sequences with bg and rg in the same position for which Invariants 1-4 hold.

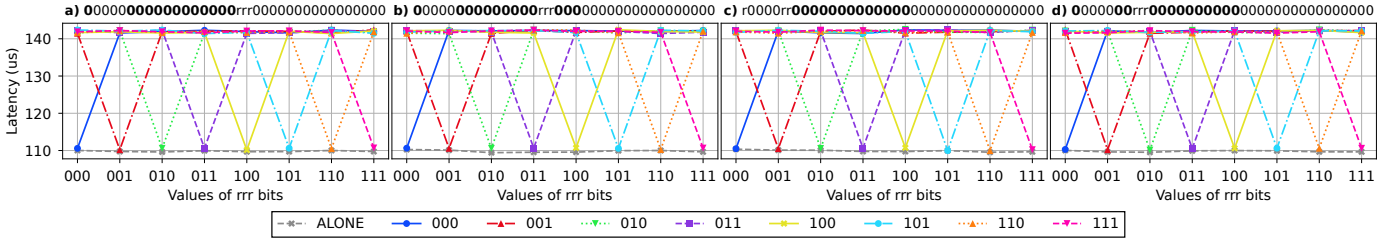

Fig. 5: Results of Phase A for which Invariants 5-8 hold. They allow verifying the address mapping tested in Fig. 4.

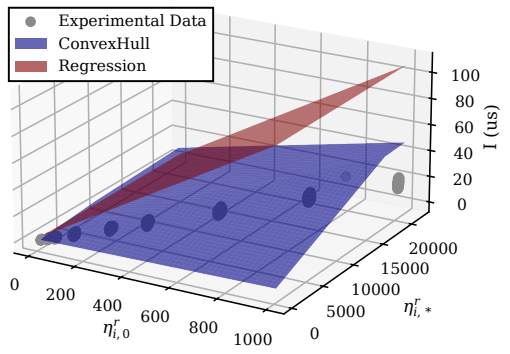

Fig. 6: Experimental results from Phase B alongside the interference bounds of the coarse-grained model.

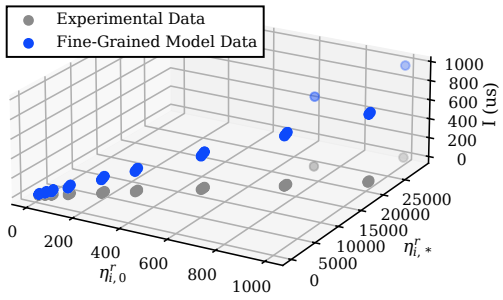

Fig. 7: Experimental results from Phase B alongside the interference bounds of the fine-grained model.

VII. Related Work

Several works studied timing bounds due to memory contention both following experimental and theoretical approaches based on the MC internals. In their work, Hassan et al. [17] presented a latency-based analysis methodology to deduce fundamental properties of the MC, including address mapping, page policy, and command arbitration schemes. Unlike our approach, which involves profiling a real-world platform, their analysis was conducted via simulation only.

Other research efforts have concentrated on providing analytical bounds to memory access times in the presence of contention. Casini et al. [10] proposed an analysis building upon rule-based MC internals to holistically determine a bound of memory access time for parallel real-time tasks. Similarly, Kim et al. [18] proposed an analysis with the ultimate goal of bounding memory interference through DRAM bank partitioning. Hassan and Pellizzoni [19], [20] and Pellizzoni et al. [21] proposed analysis methods focused on bounding the per-request delay.

Other works focused on tackling memory interference by limiting the memory bandwidth of each core in a multi-core platform. In this context, it is worth mentioning MemGuard [2], [3], which statically regulates the memory bandwidth from each core, and BWLOCK [22], that whenever a task obtains the bandwidth privilege, it enforces regulation on non-critical cores through the observation of their bandwidth consumption. Other works [23], [24] leverage the information of the memory utilization obtained through the hardware performance counters to dynamically regulate the memory bandwidth of each core. Conversely to these hardware-software mixed approaches, Zini et al. [9] explored the ARM's Memory System Resource Partitioning and Monitoring (MPAM) specification [25]. They delved into MPAM's hardware memory bandwidth regulation mechanisms and derived a memory contention analysis for each of them.

Retrieving the memory address mapping also proved to be valuable in the domain of memory allocators. In this context, Yun et al. [26] proposed PALLOC, an allocator to map virtual memory pages of different processes to different memory banks, thereby minimizing bank sharing and ensuring isolation. Other works focus on managing memory contention. Sohal et al. [27] proposed a framework to predict the timing behavior of applications in multi-core systems. They employed a profile-driven approach to provide accurate predictions. Borgioli et al. [28] presented a framework, designed for a virtualized environment, to control the memory contention

delays due to the access of shared I/O devices.

To the best of our knowledge, there have been no prior attempts at establishing a way to automatically learn MC timing models from experimental data.

VIII. Conclusion

This work presented FrATM^2, a framework to automatically learn timing models for memory contention for embedded multi-core platforms. We described the framework internals and a 3-phase workflow that allow discovering the address mapping (positions of bits in memory addresses that index banks and rows), experimentally measure memory access latency in different contention scenarios, and eventually train two MC models: a coarse-grained model based on the constrained linear regression or a convex hull, and a fine-grained model based on mathematical optimization. Our models aim to provide analytic interference bounds based on the learned experimental data. The fine-grained model achieved an accuracy of 100%, while the coarse-grained one has proven an accuracy not less than $99,97\%$ with a training time much faster than the fine-grained model, i.e., about $2s$ against the $70m$ required for the latter.

We evaluated our workflow on an AMD/Xilinx Ultrascale+ SoC (zcu102 board). We demonstrated its effectiveness by successfully discovering the address mapping of the MC and training the two models. Future work should investigate the applicability of our approach to a broader set of platforms, better investigate on other modeling methods (also based on machine learning), and extend the framework to consider memory contention generated by I/O peripherals [29].

References

[1] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, 2014.

[2] 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 *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[3] ——, "Memory bandwidth management for efficient performance isolation in multi-core platforms," *IEEE Transactions on Computers*, 2016.

[4] G. Sciangula, D. Casini, A. Biondi, C. Scordino, and M. Di Natale, "Bounding the data-delivery latency of DDS messages in real-time applications," in *35th Euromicro Conference on Real-Time Systems (ECRTS)*, 2023.

[5] P. K. Valsan and H. Yun, "MEDUSA: a predictable and high-performance DRAM controller for multicore based embedded systems," in *IEEE 3rd international conference on cyber-physical systems, networks, and applications*, 2015, pp. 86–93.

[6] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: a predictable SDRAM memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 251–256.

[7] Y. Li, B. Akesson, and K. Goossens, "Architecture and analysis of a dynamically-scheduled real-time memory controller," *Real-Time Systems*, vol. 52, pp. 675–729, 2016.

[8] JEDEC, "DDR4 SDRAM Standard JESD79-4D," 2021.

[9] M. Zini, D. Casini, and A. Biondi, "Analyzing Arm's MPAM from the perspective of time predictability," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 168–182, 2022.

[10] D. Casini, A. Biondi, G. Nelissen, and G. Buttazzo, "A holistic memory contention analysis for parallel real-time tasks under partitioned scheduling," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 239–252.

[11] P. Projects, "Jinja documentation." [Online]. Available: https://jinja.palletsprojects.com/en/3.1.x

[12] "Memory module Serial Presence Detect (SPD)." [Online]. Available: https://www.digikey.com/pdf/v/viking-technology/memory-module-serial-presence-detect

[13] P. L'ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation*, 1999.

[14] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa, "The quickhull algorithm for convex hulls," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 4, pp. 469–483, 1996.

[15] Xilinx, "Zynq UltraScale+ MPSoC data sheet: Overview," 2020. [Online]. Available: https://docs.xilinx.com/v/u/en-US/ds891-zynq-ultrascale-plus-overview

[16] Micron, "DDR4 SDRAM SODIMM MTA4ATF51264HZ - 4GB." [Online]. Available: https://media-www.micron.com/-/media/client/global/documents/products/data-sheet/modules/sodimm/ddr4/atf4c512x64hz.pdf

[17] M. Hassan, A. M. Kaushik, and H. Patel, "Reverse-engineering embedded memory controllers through latency-based analysis," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 297–306.

[18] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in COTS-based multi-core systems," in *IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 145–154.

[19] M. Hassan and R. Pellizzoni, "Bounding DRAM interference in COTS heterogeneous MPSoCs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[20] ——, "Analysis of memory-contention in heterogeneous COTS MPSoCs," in *32nd Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[21] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2010, pp. 741–746.

[22] H. Yun, W. Ali, S. Gondi, and S. Biswas, "BWLOCK: A dynamic memory access control framework for soft real-time applications on multicore platforms," *IEEE Transactions on Computers*, 2017.

[23] A. Saeed, D. Dasari, D. Ziegenbein, V. Rajasekaran, F. Rehm, M. Pressler, A. Hamann, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores," in *IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2022.

[24] A. Saeed, D. Hoornaert, D. Dasari, D. Ziegenbein, D. Mueller-Gritschneder, U. Schlichtmann, A. Gerstlauer, and R. Mancuso, "Memory latency distribution-driven regulation for temporal isolation in MPSoCs," in *35th Euromicro Conference on Real-Time Systems (ECRTS)*, 2023.

[25] "Memory System Resource Partitioning and Monitoring (MPAM), for A-profile architecture." [Online]. Available: https://developer.arm.com/documentation/ddi0598/latest

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

[27] P. Sohal, R. Tabish, U. Drepper, and R. Mancuso, "E-WarP: a system-wide framework for memory bandwidth profiling and management," in *IEEE Real-Time Systems Symposium (RTSS)*, 2020.

[28] N. Borgioli, M. Zini, D. Casini, G. Cicero, A. Biondi, and G. Buttazzo, "An I/O virtualization framework with I/O-related memory contention control for real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 11, 2022.

[29] M. Zini, G. Cicero, D. Casini, and A. Biondi, "Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms," *Software: Practice and Experience*, vol. 52, no. 5, pp. 1095–1113, 2022.