POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A JTAG-Based Memory Bandwidth Regulation Approach for Predictable Multi-Core Real-Time Systems

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Francesco Aiello, 225877

Advisor:
Prof. Marco Domenico Santambrogio

Co-advisors:
Prof. Redha Gouicem

Academic year:
2024-2025

Abstract: Real-time multi-core systems face significant challenges in ensuring predictable memory access due to shared resources like caches and DRAM. Memory unpredictability, exacerbated by inter-core interference, undermines Worst Case Execution Time (WCET) analysis, leading to pessimistic upper bounds. Memory bandwidth regulation is a promising solution to mitigate memory unpredictability in real-time multi-core systems, offering simplicity through minimal software modifications. However, its practical adoption faces challenges: regulation overhead can disrupt task execution, while aggressive frequency boosting to achieve finer-grained regulation may overwhelm the CPUs. Existing approaches often rely on heterogeneous platforms or specialized hardware to circumvent these issues, but such solutions compromise portability and scalability. This thesis introduces a novel, portable solution that leverages the JTAG debug interface, a widely available hardware feature, to regulate memory bandwidth externally, without altering the target system's hardware or software. Building on this framework, we implement one state-of-the-art regulation algorithm, MemGuard, and then propose MemAdapt, our novel regulation algorithm tailored for the framework. Experimental results demonstrate the applicability of the proposed solution on widely available but scarcely documented platforms, such as the Raspberry Pi 4 B, achieving effective regulation with minimal overhead.

Key-words: real-time system, multi-core, memory bandwidth regulation, temporal isolation

Introduction

Real-time systems are designed to process data and respond to external events within strict temporal constraints. Historically, these systems relied on single-core processors, where deterministic execution and predictable timing analysis were achievable through well-understood techniques. However, the demand for higher computational power in domains such as artificial intelligence and high-performance computing (HPC) has exposed the limitations of single-core architectures. Unfortunately, over the past two decades, the performance of single-cores has reached a plateau, as a consequence of physical constraints like power density and heat dissipation. This has led the real-time research community to ponder the idea of consolidating real-time guarantees into multi-core systems.
Commercial Off-The-Shelf (COTS) components have become increasingly attractive for real-time system designers because they deliver high performance at relatively low costs. In addition, the extensive ecosystems and

software support that accompany COTS platforms results in greatly reduced development times compared with custom-designed solutions such as ASICs or FPGA-based designs. This combination of speed, accessibility, and economic efficiency makes COTS platforms a compelling solution for next-generation real-time applications, provided that their inherent complexity does not pose an obstacle to timing constraints. In facts, COTS hardware is generally tuned for average-case throughput rather than worst-case predictability: resource contention, speculative mechanisms, and dynamic frequency scaling can cause significant latency variations that must be mitigated to satisfy the real-time constraints.
The memory hierarchy, in particular the shared Last-Level Cache (LLC) and the Dynamic Random-Access Memory (DRAM), has emerged as a primary source of unpredictability. Unlike CPUs, where scheduling policies can enforce deterministic execution, memory access times are influenced by contention from concurrent cores, complex DRAM timing constraints, and controller-level optimizations. This behavior introduces inter-core interference: a memory request from one core may be delayed due to accesses from neighboring cores, leading to an unpredictable variability in the execution times.

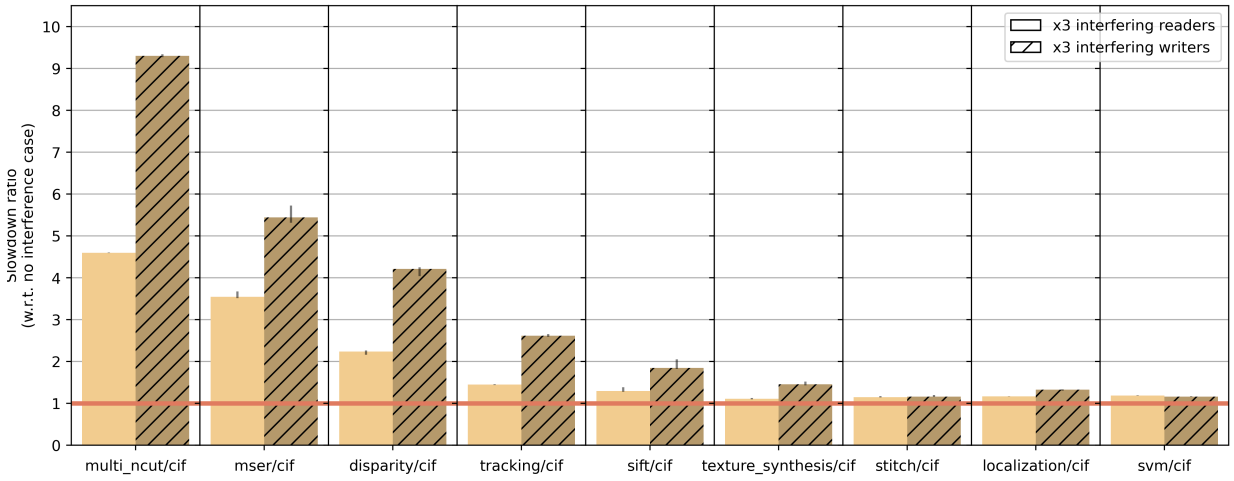


Figure 1: The effects of inter-core memory interference over co-running tasks on a Raspberry Pi 4B. The experiments employs the San Diego Visual Benchmark Suite benchmarks as foreground application, and IsolBench instances on neighboring cores to simulate the background interference. On the vertical axis we plot the observed slowdown factor, calculated as the division of the benchmark duration when in the presence of interference by the isolated execution time.

Preliminary results on a COTS platform (Raspberry Pi 4B) showed a visible effect on system performance when under heavy main memory contention. On this quad-core platform co-running tasks suffered a slowdown up to a factor of 9.3 with respect to isolated execution, as shown in Figure 1.
Several attempts to cope with memory hierarchy unpredictability exist, both in the hardware and in the software domain. At the hardware level, custom-designed memory hierarchies (e.g., [23]) have been proposed to enforce strict bandwidth guarantees or latency isolation. However, these solutions require significant micro-architectural changes, rendering them incompatible with Commercial Off-The-Shelf (COTS) platforms. On the software side, OS or hypervisor-based mechanisms (e.g., [27], [17]) attempt to regulate memory bandwidth through periodic monitoring and core throttling. While portable, heavy regulation overheads limit the effectiveness of such approaches.
A middle ground is occupied by hybrid solutions like MemPol [30] and MemCoRe [10], which exploit platform heterogeneity, such as on-chip FPGA integration, to achieve fine-grained regulation without altering the flow of execution. These systems are based on tight cooperation between software and hardware functionalities to decouple the regulation logic from the regulated tasks. However, their applicability remains constrained to specialized platforms (e.g., Zynq UltraScale+), limiting their use in mainstream embedded platforms.
This thesis advances the state of the art by introducing a JTAG-based memory bandwidth regulation framework that overcomes the limitations of prior approaches. Unlike hardware-dependent solutions, our method requires no micro-architectural modifications, and unlike FPGA-centric frameworks like MemCoRe, it operates on any processor supporting the widely available JTAG debug interface. By exploiting external debug capabilities, we enable interference-free memory regulation without altering the system software or hardware. This work bridges the gap between regulation effectiveness and platform availability, offering a portable, COTS-compatible solution for memory management in multi-core real-time systems.
We designed MemAdapt, a dynamic memory bandwidth regulation mechanism to complement the JTAG-based regulation. MemAdapt improves the previous state-of-the-art algorithms by providing a versatile and more

robust resource allocation, able to adapt to many platforms with different requirements.
We structure the rest of this thesis as follows:

1. In Chapter 1, we provide some background about hardware-assisted debugging, clarifying the tasks of a physical debugger and how the most common tasks are handled.
2. In Chapter 2, we review the major sources of unpredictability of memory hierarchies;
3. In Chapter 3, we give a bird's eye view of the state of the art in the context of memory unpredictability mitigation.
4. In Chapter 4, we present our contributions in the field, from the JTAG-based approach to the algorithmic details of MemAdapt.
5. In Chapter 5, we evaluate the effectiveness of our proposal and prove that we reached the goal of achieving memory isolation.

1. External debugging

Debugging is an essential part of software development, as it ensures that the final application is correct and coherent with the specifications. Applications designed to work in embedded environments face further challenges in this regard, mostly due to (1) the decoupling of the platform where the software is produced from the target platform where the final application is designed run, and (2) the limited access to the target platform. To overcome such difficulties, a viable solution is to exploit the hardware debug capabilities of the target platform. This is one of the most common options in the embedded domain, as it is relatively inexpensive and widely available.
The particular configuration and the offered features can vary among the different vendors and platforms, but we can try to outline the structure that they usually follow:

- Physical connection: that is, how the connection with the target platform is done; this is widely standardized and the most notable examples are JTAG, SWD, or USB.
- Debug bridge: an intermediate component that allows the user to interface with the different internal debug modules.
- Debug module: the internal component implementing the debug functionalities, for instance, by providing access to the system registers or the internal buses.

From now on, we will take as an example one specific implementation, the Arm CoreSight architecture and the JTAG standard, that will lay the foundation for the methods presented in the next chapters. For simplicity, we will refer to the debug target platform as *target* and the platform performing the debug as *debugger*.
The reference for this architecture is the ARM Debug Interface v5 Architecture Specification [1], while the JTAG interface is standardized under the IEEE 1149.1 standard [8].

1.1. The JTAG interface

The JTAG is a set of standards for verifying and debugging printed circuit boards (PCBs). It is named after the group that proposed it, the *Joint Test Action Group*, which played a key role in developing the first JTAG standard, the IEEE 1149.1, in 1990.
While initially designed for board-level testing, JTAG has evolved into a versatile tool for advanced applications, including embedded debugging, firmware updates, and diagnostics.
We will only cover the details of the Test Access Port (TAP) specification, which is part of the IEEE 1149.1 and used as a physical access point to debug circuitry.
Access to the TAP is carried out through the following physical connections:

- Test clock input (TCK),
- Test mode select (TMS),
- Test data input (TDI),
- Test data output (TDO),
- optionally, Test reset input (TRST).

The TAP logic is implemented inside a TAP controller, whose behavior is standard and defined by the finite-state machine shown in Figure 2. A state change of the TAP controller is driven by the TCK and TMS values: as soon as the TCK signals a transition from a low to a high value, the state of the controller is updated depending on the value of TMS. Therefore, every state in the finite-state machine shows two outgoing transitions, one corresponding to a TMS low value and the other to a TMS high value.
We further remark that the test logic is asynchronous and does not rely on the target or debugger system clocks, but only on signal transitions. This means that the TCK signal should not necessarily be periodic and that the maximum speed of the circuit depends on:

- the frequency at which the debugger can drive the signals, and

- the electrical constraints on the physical connection and within the TAP controller.

The optional TRST connection is used to bring the TAP to a reset state. Not only does this bring the state machine into the Test-Logic-Reset, but it also resets the state of the connected components, such as an Arm DAP. If the TRST is implemented, then the logic stays in reset state as long as the TRST is kept low; during normal operation, TRST should then be asserted high.

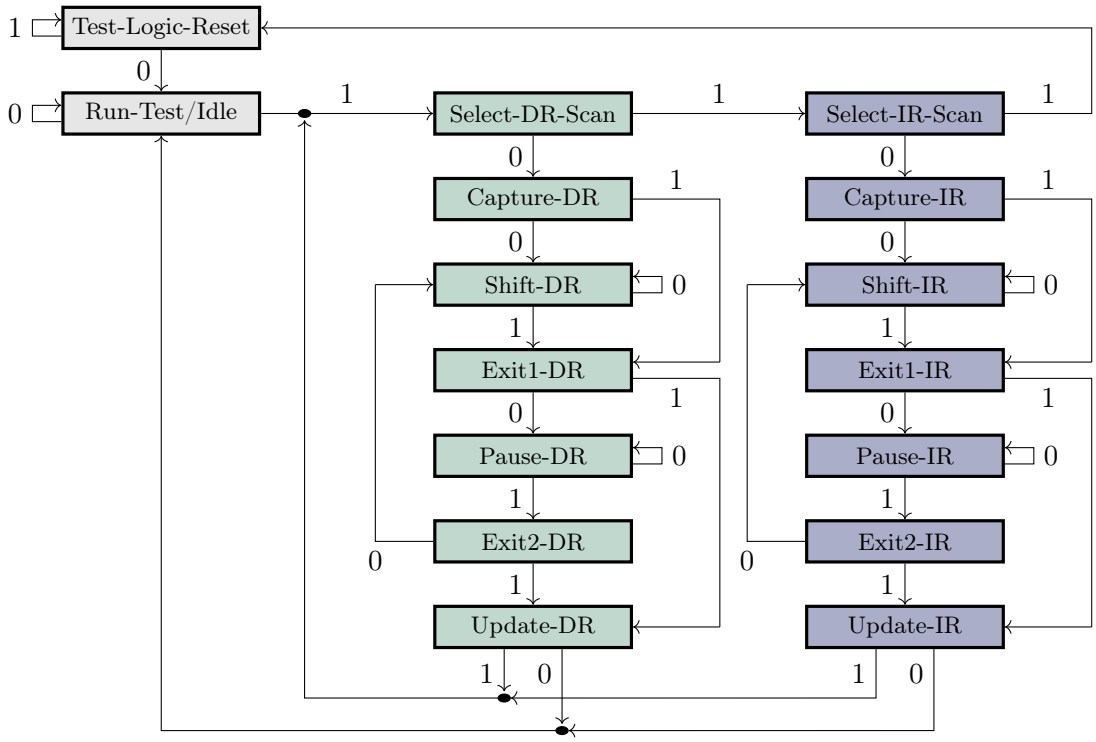


Figure 2: The TAP finite-state machine representation. The annotations on state transitions indicate the value of the TMS pin that drives such transition.

The TDI and TDO are used to send and receive data to the TAP: when the state machine is in a Shift state (`Shift-DR` and `Shift-IR` in Figure 2), TDI and TDO are connected through a specific shift register, whose size depends on the size of the register we want to write to. The procedure to manipulate data in a register can be summarized as follows:

- When the TAP enters the Capture state, the register value is read and copied into the shift register between TDI and TDO
- When in the Shift state, on a falling edge of TCK, the least significant bit of the shift register is written to TDO, on a rising edge of TCK, the shift register is shifted by 1 bit, and the value read on the TDI is inserted as the most significant bit.
- When the Update state is reached, the value of the shift register is finally copied back to the original location.

As we can see in the state machine, only two kinds of accesses can be made, to the Instruction Register (IR) and the Data Register (DR). The two registers must be operated in cooperation: we instruct the IR with the specific operation we want to perform, and then we access the DR to send and receive the desired data.
We now illustrate as an example the sequence of operations required to read the TAP identification code (IDCODE). We describe the sequence in terms of TAP state machine transitions, where intermediate steps are omitted for simplicity.

1. The TAP state machine enters the Capture-IR state: since the IR is, in general, a write-only register, the shift register is pre-filled with an arbitrary value.
2. The TAP state machine enters the Shift-IR state: we discard the read value and shift in the instruction to read the id code (whose value depends on the specific TAP implementation) one bit at a time.
3. The TAP state machine enters the Update-IR state: the instruction previously written takes effect, and the DR shift register is now connected to the id code register.
4. The TAP state machine enters the Capture-DR state: the value of the id code register is copied into the shift register.
5. The TAP state machine enters the Shift-DR state: we stay in this state until the id code is completely shifted towards TDO. Since the id code register is read-only, the value shifted in from TDI will not have any effect.

6. The TAP state machine enters the Update-DR state: no update takes place in this state.
7. We leave the TAP controller in the Run-Test/Idle state so that the TAP is ready to receive further instructions.

Multiple TAPs on the same board are allowed, eventually sharing some external connections. The standard specifies different ways in which TAPS can be interconnected, even though the most common choice is to connect them in daisy-chain.
When multiple TAPs are connected in this fashion, the external interface appears the same, but the internal connections are different: the external TDI is connected to the first TAP, while the first TAP's TDO is connected to the second TAP's TDI, and so on until the last TAP's TDO is exposed externally. The interconnection of all the shift registers in the TAPs is commonly referred to as *scan chain*. The TCK and TMS signals are shared among the TAPs and, therefore, all TAP controllers are always in the same state. To change the current instruction in each TAP, all the instruction codes, one after the other, should be shifted in the chain. Then, the same approach is used to read and write a value into the selected DRs. With this configuration, it is not possible to singularly disable the TAPs, in case we want to operate on a single TAP.
As a countermeasure, we can exploit the BYPASS IR instruction to connect the unused TAPs' DRs to a side-effect-free 1-bit shift register. This has the advantage of speeding up transactions, as fewer bits are needed to be "put" on the chain, and protecting the unused TAPs from unwanted writes.

1.2. Arm DAP

The Arm Debug Access Port (DAP) is the Arm extension of the IEEE 1149.1 TAP, it is specified by the Arm debug interface interface[1] (ADI) and is part of the CoreSight architecture. It is constituted by Debug ports (DP) and Access ports (AP), hence the name DAP.
Each Debug port manages the low-level physical connection to the DAP; they can be a SW-DP, using a serial wire debug (SWD) protocol, or a JTAG-DP, implementing the IEEE 1149.1 standard.
Access Ports are used to group higher-level debug capabilities, like a Memory Access Port (MEM-AP), to access the CPU debug registers, or a JTAG-AP to access a JTAG boundary scan registers.
In the rest of this chapter, we will only focus on the JTAG-DP and MEM-AP, which will lay the groundwork for the rest of this work.
The JTAG-DP fixes the IR length at 4 bits and extends the IEEE 1149.1 TAP by adding the following IR instructions:

- DPACC, to configure the Debug Port,
- APACC, to access registers in an Access Port, and
- ABORT, to force an AP abort.

The numerical value of the IR instructions and the length of the respective DRs are all specified by the standard.
Accessing multiple DP and AP registers using only two IR instruction is made possible by using an additional level of indirection. For both the DPACC and APACC, the respective DR is 35 bits long. When accessing this DR, the value shifted into the scan chain should be formatted as follows:

- the most significant 32 bits are used for transmitting the data to write;
- the following 2 bits are used for addressing either an AP or DP register, inside a bank of 4;
- the least significant bit is used to initiate a read transaction or a write transaction.

The value shifted out of the DR follows instead the structure:

- The most significant 32 bits are used for retrieving read data,
- The least significant 3 bits represent an acknowledgment from the Debug or Access port.

Therefore, reading and writing cannot take place within the same DR access, but requires more steps:

- The first step is always to write to the DR, regardless if this is a read or write. If this is a read, we need to start a read transaction by adjusting the least significant bit and indicate the target register using the next 2 bits; if this is a write, we also need to provide the value to be written in the 32-bit field.
- Then, the DR should be accessed again, and we analyze the value shifted out from the scan chain. The acknowledgment field signals whether the previous transaction has been completed; if not, we repeat the access until we read a positive acknowledgment. If the previous transaction was a read, we also find the read data in the 32-bit field.

We hereby highlight the inefficiency of the previous algorithm: in the first access, we discard the value shifted out of the register, while in the second access, we forget about the value shifted in. This opens up for optimization: we can chain sequential transactions in such a way as to interlock requests and responses. For instance, to perform two sequential reads:

- We write to the DR to start a read transaction to the desired DP or AP register.

[1] This document is based on the Arm debug interface version 5; the newly released version 6 will not be covered, as our work is mainly based on version 5.

- We write to the DR to start the second read and, at the same time, we shift out of the DR the acknowledgment and the read data. If the acknowledgment doesn't mark the transaction as completed, we repeat the access with the same format. It is important to notice that if the acknowledgment is negative, the second read transaction is always rejected.
- The last step is to access the DR and retrieve the second read value. The same considerations for the acknowledgment hold.

The Debug port configuration registers can be accessed using the 2 bits in the address field, as there is a simple one-to-one mapping that allows to choose from:

- Control/Status (CTRL/STAT) register,
- AP Select (SELECT) register,
- Read Buffer (RDBUF) register.

The Control/Status register, as the name suggests, provides control and status information about the access port. It has many uses, but the most common operations are to power up the debug domain and check for errors in the Access Port.
The Read Buffer is a register that has no effect: it is used when chaining multiple reads with the previously mentioned method in such a way that we can switch to the Read Buffer before fetching the last returned read value, so that we can read the value without initiating any further transaction in the Debug Port.
The AP Select register is used in conjunction with the APACC instruction. This is because the number of registers in an access port is larger than 4, and we cannot address them all using only 2 bits. For this reason, the Access Port is organized into banks of 4 registers, and the 2 bits of the 35-bit DR word are used to select a single register in the bank. To switch bank, we need to access the Debug Port (DPACC) and write to the AP Select the bank we want to select. Since we can have multiple Access Ports, the Select register is also used to select which Access Port we want to access.
Now that we know how to handle the Debug Port and perform accesses to the Access Port, we can present the Memory Access Port (MEM-AP).

1.2.1 Memory Access Port

The Memory Access Port provides access to a set of resources using a memory-mapped interface. In the simplest case, such resources are the debug registers of an Arm Cortex core, but it can also be used to perform arbitrary accesses in main memory.
The registers we will most likely want to access in this Access Port are:

- The Identification Register (IDR), which identifies the Memory Access Port, and suggests to which bus it is connected to;
- The Debug Base Address (BASE) address, which stores the address of the ROM table and provides a good starting point to understand the addressing space;
- The Control/Status Word (CSW) register, for configuring the access to the memory system;
- The Transfer Address (TAR) register, which stores the address where we want to access;
- The Data Read/Write (DRW) register, which is used as a buffer for reads and writes;
- The Banked Data (BDn) registers, for optimizing access to sequential addresses.

The usual workflow for interacting with the MEM-AP starts by finding the correct AP inside the DAP. By reading the respective IDRs, we can finally identify if the selected AP is a Memory Access Port and the bus type it grants access to. For Arm architectures, memory accesses usually pass through a debug bus (APB) or the main system bus (AHB/AXI). A high-level depiction of the bus interconnections is shown in Figure 3.
If needed, we can read the address of the ROM table from the BASE register. This is indeed useful, as the ROM table lists the available debug components and their respective base address .
To perform the actual access in the connected memory, we start by configuring the CSW, for instance by tuning the access width to match the target register size; then, we write in the TAR register the memory address we want to read or write from. Any access to the DRW register will now behave like an access to the chosen memory location: writing to the DRW will also write to the chosen address, while reading to the DRW will read the chosen memory address.
The Banked Data registers are similar to the DRW but allow access to successive memory locations starting from the address in TAR. This makes it possible to access 4 sequential 32-bit words without updating the value in the TAR for each access.
If an error occurs in a memory transaction, it will be reported in the CTRL/STAT register in the Debug port. This bit is called the Sticky Error bit, because it will stay set until it is manually cleared. The reason for this is to allow the user to check for errors only once at the end of a sequence of transactions.

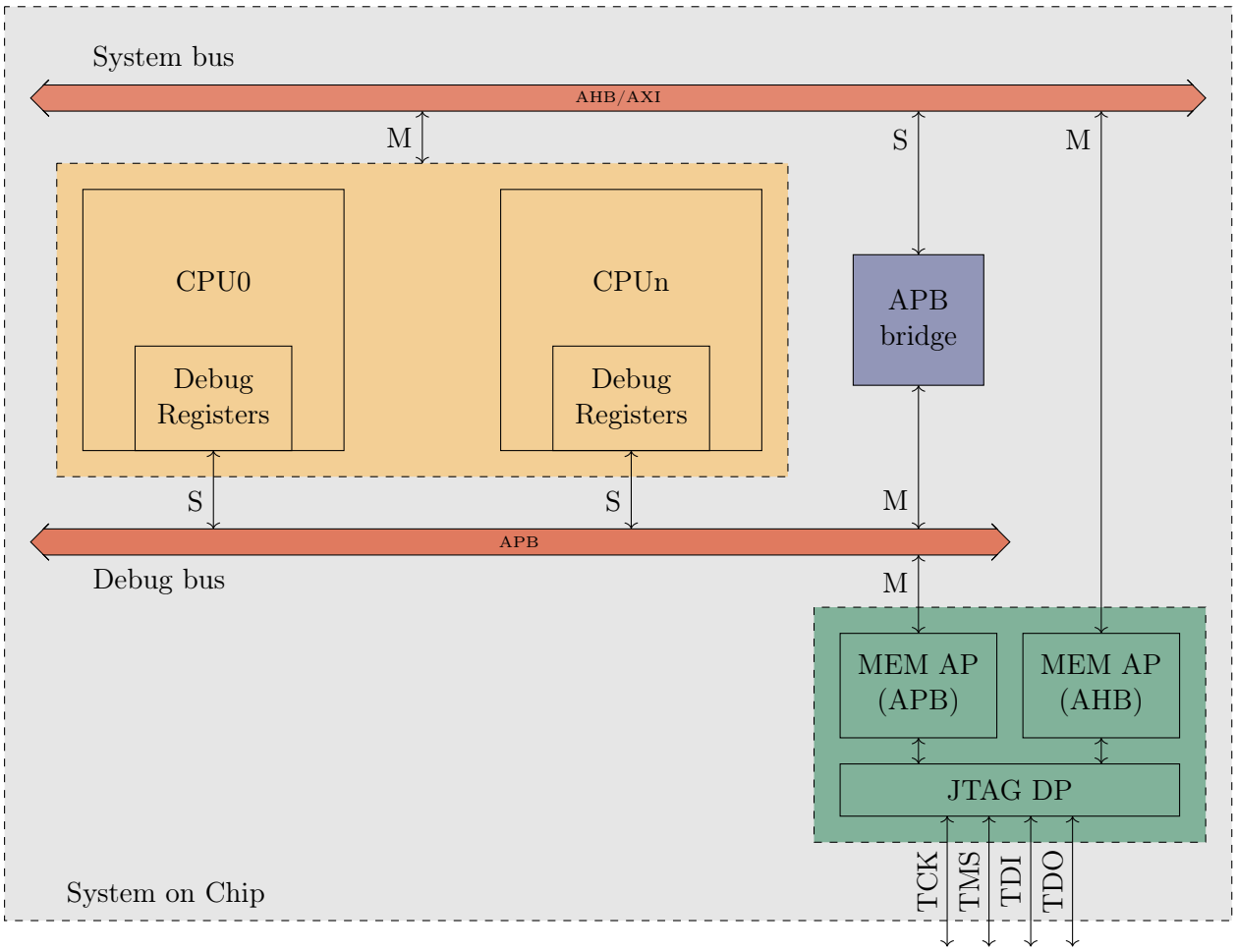


Figure 3: A simplified block diagram of the MEM-AP bus connections. The arrows show the connection between buses and components, where an "S" marks a slave on the bus and an "M" a master.

1.3. System debug registers

Up to now, we have seen how the debug resources can be accessed from the outside, but we haven't delved yet into the details of how debugging is concretely done. This is also the part that is most dependent on the underlying architecture. At the time of writing, version 5 of the Arm ADI is still prevalent among Arm Cortex CPUs, regardless of the architecture profile. Once we reach the CPU registers, the rules become stricter, and the specific model of the CPU plays a crucial role. Unless specified, we will cover the case of an *Arm v8.0-A* architecture.
For our use case, the modules that will have a major impact are:

- the Cortex general Debug,
- the Performance Monitors,
- the Cross-Trigger Interface.

Latest architectures may provide even further debug features, such as hardware tracing through the Embedded Trace Macrocell and several extensions to the Performance Monitors.

1.3.1 The Cortex-Debug registers

The Cortex debug registers allow us to perform all the basic debug operations, like setting breakpoints, halting execution, resuming execution and reading the CPU registers.
Halting a CPU is a very common operation in debugging: whether we want to read the CPU registers or acquire other data, it's always a good idea to temporarily stop execution during these operations, as it guarantees the coherency of the read results.
For Arm CPUs, halting not only suspends the execution but also puts the core into a debug state. In this state, the CPU stops fetching newer instructions from memory, waiting for the debugger to take control of the execution. Special registers exist to pass arbitrary instructions and data to and from the CPU: they are the Instruction Transfer Register (ITR) and the Data Transfer Register (DTR).

Several events may cause the system to enter debug mode, such as:

- a breakpoint or a watchpoint has been reached,
- the current running program has executed a `Halt` instruction,
- an external debug request is pending,
- the CPU is in Halting Step mode, and a single instruction has completed since the core has resumed from debug.

The default behavior is to enter debug precisely: this process is very similar to taking a precise interrupt and, therefore, we need to take into account that the CPU will not halt instantly. It is possible to enable imprecise entry to debug by converting an already existing pending debug request into imprecise. Since, in general, accessing the system registers from the external debug interface is slower than the overhead to enter debug, this feature has a limited impact. According to the manuals, the main use case is to continue execution in case the memory system is not responding and all memory accesses are stalling.
When the CPU is finally halted, the debugger can make use of the ITR and DTR to communicate with the core. The ITR is used by the debugger to issue instructions to the core. A limited set of instructions can be executed while in debug state, mostly load/store operations: this is because the use case for this is to retrieve useful data from the current execution context and pass it over to the debugger. The DTR is a simple buffer that can be accessed from both the CPU and the debugger that can be used for data transferring.
Suppose, for instance, that the debugger wants to retrieve the value at the (virtual) address `0xdeadbeef`, a possible sequence of operations to accomplish this task can be:

1. Write to the ITR to instruct the CPU to store a register, say Rn, to the DTR: this motivation for this step is that we will need to use a register as a buffer and, by passing the current value to the debugger, we will be able to restore the original value at the end of the operations. The value to be written to the ITR is the specific binary encoding of the required instruction (e.g., `msr` or `mcr`, depending on the target architecture).
2. Read the DTR value and save for later the value of Rn.
3. Write to the DTR the value `0xdeadbeef`.
4. Write to the ITR to instruct the CPU to copy the value of the DTR into Rn (e.g., `mrs` or `mrc`, depending on the target architecture).
5. Write to the ITR to instruct the CPU to load the value of the location `0xdeadbeef` into Rn (e.g., `ldr Rn, [Rn]`).
6. Write to the ITR to instruct the CPU to copy the value of Rn into the DTR.
7. Read the value in the DTR, which now contains the desired value.
8. Write the original value of Rn into DTR.
9. Write to the ITR to instruct the CPU to copy the value of the DTR into Rn.

If all the operations are executed correctly, the CPU should be in the same state as before and the debugger should have fetched the value at location `0xdeadbeef`.
To speed up the transferring of sequential data, the CPU implements a so-called *Memory Access mode* that automatically performs all the previous operations each time the DTR is accessed.

1.3.2 The Cross-Trigger Interface

The Cross-Trigger subsystem is used for sending and receiving signals between different debug components. The Cross-Trigger Interface (CTI) allows connection of input and output triggers with channels: input triggers cause a pulse on the chosen channel, and a pulse on the channel causes the output triggers to fire. Multiple inputs and outputs can be connected to the same channel, and each channel can be private to a single CPU or connected with the same channel of all the other CPUs.
For our use case, the relevant triggers will be:

- The Performance Counters Overflow signal, as input trigger;
- The Debug request signal, as output trigger;
- The Debug restart signal, as output trigger.

For manually halting or resuming the core, it is required to choose a channel and connect the respective output trigger to that channel. To fire the trigger, it is possible to manually cause a pulse on such channel by writing into the appropriate system register.

1.3.3 The Performance Monitoring Unit

The Performance Counters cover a wide range of use cases: they are extensively used from both the external debug interface and from the software running on the cores. A common example is the Linux Perf tool, which analyses the performance of programs using the hardware capabilities.

Arm provides an extensive set of events that can be implemented, and more platform-specific events can be added by individual manufacturers. However, most implementations do not include the full list due to practical design constraints, offering only the events required for basic debug functions.
Some examples are the cache counters, which count the number of refills or write-backs at each cache level, or the cycle counter that unconditionally increases at each CPU cycle. Some extensions to the Arm-defined events can be found in some Memory Controllers, offering more specific counters that enable fine-grained measurements of memory accesses.
When a Performance Counter overflows, the default behavior is to start again from 0 and continue counting the event. The overflows generate a signal that can be routed through the CTI or forwarded to an Interrupt Controller to generate an interrupt request.
The counters can be realized with 32-bit or 64-bit registers, depending on the architecture. To count larger amounts, it is possible to chain multiple counters together so that when one counter overflows, the other one is incremented.

2. Memory unpredictability

The memory subsystem is organized in a layered structure, with each level tuned to achieve a specific balance between speed and capacity, in order to mitigate the differences between the rapid processor execution times and the slower retrieval of data from the main memory. The intermediate layers are usually constituted by caches, ranging from small and very fast caches to larger but slower caches.
In general purpose processors, the design choice is usually to improve the performance in the average case scenario, by optimizing common operations, such as sequential accesses, and exploiting memory-level parallelism. Throughout the years, many of these features have been introduced [18], bringing noticeable benefits performance-wise, but at the cost of adding complexity to the final infrastructure.
Making a precise estimation of the time required to access a specific memory location is indeed a challenging problem. In this chapter, we will briefly review some common patterns in memory hierarchies, and try to understand the reasons that hinder such a study. We will organize the content by component, following the memory transaction's journey from its initiation in the CPU to its completion at the main memory, highlighting at each step the main sources of unpredictability.

2.1. Issuing a memory instruction

Any computing system that follows the Von Neumann architecture requires a memory. It stores the description (e.g., code) of the programs to execute and is eventually used by the program during its execution. We consequently expect that, sooner or later, any program will have to perform a memory access.
How each memory access is handled is a choice made by the designer of the CPU. We will not delve into the details of the single implementations, but we remark that any modern architecture tailored for general-purpose computing is very likely to implement some degree of parallelism already at CPU level.
The major distinction we can make is between in-order and out-of-order architectures. Out-of-order architectures do not guarantee that memory accesses are completed in the same order they are issued, and multiple Load/Store units can allow multiple accesses to take place at the same time.
So, we have a variable number of CPUs, accessing memory to fetch instructions and accessing data, eventually supporting multiple outstanding transactions simultaneously.

2.2. Intermediate caches

Caches come in variable numbers, usually organized in different levels. Most cache hierarchies are made up of two or three levels, but more levels are not uncommon in systems that must hoard large amounts of data.
Caches are usually handled transparently by the hardware: the location in the cache is calculated starting from the target memory address. For each address, a Set is computed from a subset of the address bits and a Way is heuristically computed; the Set and Way couple determines the physical location of the cache line to be allocated. The replacement algorithm for computing a Way may vary on the implementation: it may replace the last used cache line, use a round-robin approach, or choose randomly.
We can already spot another source of unpredictability, as computing whether a block of data will fit or not in the cache requires predicting the replacement policy and the current state of the cache.
In a multi-programmed environment, we also need to take into consideration that the cache may be shared among the different tasks executing on a single CPU, and concurrent tasks may see their allocated cache lines evicted by each other.

The contention problem is further exacerbated in the scenario of a shared cache, where multiple CPUs are competing for the available cache lines.
Caches implement parallelism in different ways, such as:

- allowing parallel access to cache lines,
- allowing multiple misses to be pending at the same time.

Parallel access to cache lines is provided by physical subdivision into different banks; that is, each bank can operate in parallel to other banks. This subdivision usually depends on the address associated with the cache line; in the simplest case, a subset of the address bits is used to select the bank. Eventually, the partitioning can be made at a finer granularity than a cache line, in order to split a single cache line into different banks.
When a miss in cache occurs, the request is forwarded to the next level in the memory hierarchy. In the simplest case, the cache behaves in a blocking way, suspending all other accesses until the miss is resolved. Conversely, some caches implement special buffers for queuing pending miss requests, usually referred to as Miss Status Holding Registers (MSHR), and for queuing pending write-backs, referred to as WriteBack Buffers. In this way, the cache can still function while refills and write-backs are still pending.
To wrap everything together, when the cache is accessed:

- The cache bank and a cache set are computed;
- We wait for the cache bank to be available;
- When the cache bank is available, we can determine if the access is a hit or a miss;
- If we have a miss, we need to forward the request to the next level in the memory hierarchy;
- If we have available MSHR or WriteBack buffers, the cache can continue to serve other requests; otherwise, it blocks.

2.3. Main memory

When the access misses in all caches, the operation will move on to the main memory. With respect to the caches, the main memory appears to be sensibly slower. The technology with which it is built is indeed different, as we are moving from an SRAM to a DRAM memory. The main memory may be the performance bottleneck for memory-bound applications, and a failure to provide a reliable service will have a noticeable impact on the performance. As a result, contention of main memory may have serious consequences for the execution of the running applications.
To better understand the sources of contention, we briefly review the internal organization of a DRAM.
The CPU communicates with the main memory using the Memory controller, through one or more channels. Each channel interfaces with one or more memory modules, which contain multiple DRAM chips organized into ranks. In its turn, each DRAM chip is made up of multiple banks.
A bank is organized into rows and columns, creating a 2-dimensional array of memory cells. To realize the data access, the specific row is selected and copied into a row buffer; all the operations take place in the buffer, which is eventually written back to the original row.
The reason behind this architecture is again parallelism, since every bank can operate independently from the others.
The memory controller decides how to link a memory address to a physical location in main memory. This is a critical decision, as it determines the bank parallelism that we can extract. For the principle of spatial locality, the most common choice involves:

- allocating consecutive pages to different banks, to parallelize sequential access in memory;
- allocating consecutive bytes to the same row, to limit the number of row accesses when accessing a chunk of data;

The final product of this decision is usually a linear function that assigns to each address: a channel, a rank, a bank, a row and a column. The channel, the rank and the bank are usually chosen from the bits near the 12th and 21st[2], since they correspond to the least significant bit of a page frame of size 4 kB and 2 MB size. From the unused bits, the least significant bits usually select the column and the remaining ones select the row.
The memory controller stores the sequence of DRAM operations into per-bank queues and orchestrates the scheduling of the bank-specific queues. This usually comprises:

- to optimize operations in each bank queue, and
- to exploit bank-level parallelism.

Within each bank queue, requests are sorted to reduce row conflicts, allowing newer requests to "skip the queue" if they operate on the currently selected row. The algorithm, sometimes referred to as First-Ready First-Come-First-Serve (FR-FCFS), is a potential source for priority inversion issues.

[2]Conventionally assuming that bit 0 represents the least significant bit in an address.

The next choice is to choose a request from the multiple bank queues. This is usually done using a Round-Robin (RR) policy to improve the bank parallelism: DRAM delays are in fact smaller if the next operation is targeting a different bank.

3. State of the Art

Several attempts [7, 11, 29] to perform a detailed WCET study, taking into consideration memory interferences, exist in literature. Such methods rely on complex formulas and usually end up overestimating the actual WCET. The main issues with these approaches are due to the inherent pessimism in (1) measuring the memory footprint of applications, and (2) computing the worst-case interference from co-running tasks. The approach in [7] successfully proposes a linear programming problem, solvable in polynomial time, to compute the worst-case delay, given an upper bound for the memory accesses of the tasks. While we greatly value the effort taken into these works, we also underline their limited applicability in a real scenario, as they tend to oversimplify the underlying architecture and their pessimistic results will lead to an under-utilization of the system under study. The difficulties arisen from the previous methods have led the research community to look for alternative solutions, where an active management of the memory takes place in an attempt to reduce its intrinsic unpredictability. This opens up new directions in memory management, most prominently resource partitioning and memory bandwidth regulation.

3.1. Memory bank partitioning

An intuitive solution to cope with the problem of resource contention is by employing some sort of partitioning scheme. For DRAM bank contention, the state-of-the-art software solutions [11, 12, 22, 28] agree on the usage of bank partitioning through page coloring.
Page coloring refers to a method for allocating pages according to a chosen "color", that is usually given by a subset of the bits from the Page Frame Number. It is relatively easy to implement and has been known for many years [26]. Bank partitioning can be achieved through page coloring by choosing the colors such that each one maps to a different region of memory residing within specific memory banks. Albeit its simplicity, many shortcomings affect this method, such as:

- Implicit partitioning of main memory, which may become an issue if many partitions are instantiated;
- Implicit partitioning of caches, if the coloring bits overlap with the cache set index bits;
- The allocated pages are usually non-contiguous, slowing down certain operations, such as DMA transfers;
- Results in large overheads if the partitions are changed at run-time, as it requires relocating all the pages in memory.

To partition the DRAM banks, it is first required to determine how the memory controller maps each physical address to a specific bank. Sometimes, this information is publicly disclosed by the SoC vendors, while other times, the mapping should be obtained by reverse-engineering the memory controller. Many techniques exist [3, 14] for this, but the reliability of such results, especially when using software-only methods, highly depends on the complexity of the memory controller.

3.2. Memory bandwidth regulation

Memory bandwidth regulation techniques solve the problem of the detailed hardware assumptions by treating the memory subsystem as a black box: we consider the memory subsystem as a server that distributes a variable bandwidth to the CPUs, as shown in Figure 4. Under full usage, the total bandwidth provided can vary within a certain minimum and maximum value, which we refer to as Guaranteed bandwidth and Peak bandwidth. Under normal usage, we assume that the serviceable bandwidth is non-deterministically distributed to clients (i.e., the CPUs), without making any further assumption on the access pattern and the memory controller behavior.
The problem of memory unpredictability is captured by this model through the unpredictable redistribution of the serviceable bandwidth among the users. Memory bandwidth regulation techniques focus on this point, acting on bandwidth redistribution to provide a lower bound guarantee for the memory bandwidth.
Monitoring the currently used memory bandwidth is a topic that requires special care. We want, first of all, to be able to measure the guaranteed bandwidth, and, later on, to measure the currently used bandwidth for correctly enforcing the regulation. As shown in Chapter 1, the Arm PMU provides good tools for this use case, but Intel CPUs also provide similar features [9]. The cache miss counters can be a viable solution, as they are widely available on many platforms, often being able to distinguish a cache line refill from a cache line write-back. Even though they are not as pervasive as cache miss counters, other means of performance analysis may be present in the system under study. There can be performance counters located within the Memory

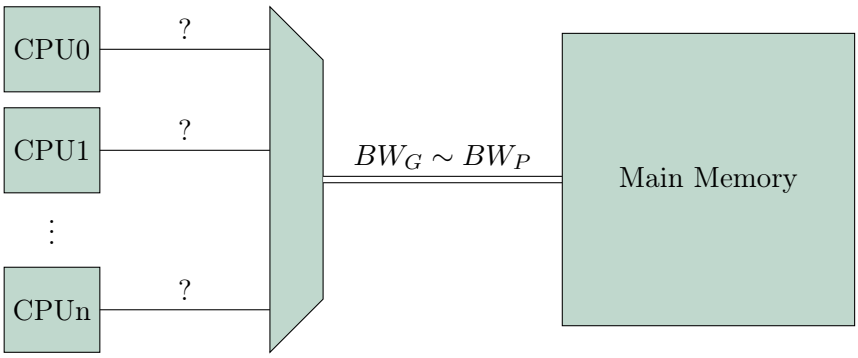


Figure 4: The adopted memory model used in memory bandwidth regulation. Under full utilization, the aggregated memory bandwidth from the cores varies between the Guaranteed Bandwidth (BW_G) and the Peak Bandwidth (BW_P). How the bandwidth is distributed among the cores remains unpredictable.

Controller that allow for a more precise estimation of memory usage [17], even though they are, unfortunately, rather uncommon. More creative solutions have exploited the hardware support, like in CPU+FPGA platforms, to probe transactions directly on the system bus [15, 16]. This is by far the most accurate measurement that can be achieved, but comes at the price of losing information about the originating CPU, because the bus transactions are issued from the LLC, which acts as a single master for all the CPUs.
The Guaranteed bandwidth and, eventually, the Peak bandwidth, should be computed with the chosen memory footprint analysis framework. It is, in general, a very challenging task to compute these values and at the same time keep the assumption of the black box memory model. The most common choices in literature usually involve a trade-off between accurate measurements and converting the black box model into a white box one.
The last missing block for building a memory regulation framework is the active bandwidth enforcement. Given that, in general, we cannot manipulate how the memory redistributes the available bandwidth to the CPUs, the usual approach is to forbid access to the memory from the CPU. This is possible, for instance, by scheduling a high-priority CPU-intensive task on the CPU [4, 27], or halting execution using debug facilities [10, 30]. The downside of the latter is that nothing can run on the CPU while it is halted, including the software for bandwidth regulation; but, on the other hand, they incur smaller halting delays and provide better isolation.
Alternatively, some hardware-specific options may be available, like the QoS extensions [2] for the Arm CoreLink NIC-301 Network Interconnect, that can be used to regulate traffic at the interconnect level. By configuring the transfer rate, burst size and peak rate, it is possible to enforce a specific bandwidth for memory transactions. Unfortunately, this approach does not allow the distinction of traffic from different CPUs, but it can, differently from other approaches, regulate the traffic from accelerators. Works like [19] successfully leveraged this feature to enforce system-wide memory regulation. A similar feature is adopted by Intel with the Memory Bandwidth Allocation (MBA) and Memory Bandwidth Monitoring (MBM), with the difference that they allow per-core memory bandwidth control [6].

3.2.1 State of the art implementations

The pioneer of memory bandwidth regulation is MemGuard [27], a purely software implementation that is based on a very simple approach: each core is assigned a memory budget to use over a fixed period and, as soon as the budget is exceeded, the execution is suspended until the next period. The algorithm runs in two different moments:

- periodically, to resume all cores and
- when a core's budget is exceeded, to suspend execution.

If the total memory budget corresponds to the Guaranteed memory bandwidth and each core is assigned a fraction of it, we can conclude that, under full memory utilization, in the worst case, inter-core interference can not cause a core to use less budget than the allocated one.
Alongside the basic features, several improvements were also proposed, in an effort to provide a better average case bandwidth, while preserving the worst case.
The simplest improvement regards the scenario where all cores are suspended because they exceeded their budget: keeping all cores suspended does not provide any benefit. We can therefore resume all cores for execution until the next regulation period. This has the high-level behavior of preserving the worst-case memory bandwidth, as the allocated budget was already exceeded, but improving the overall average performance.
Another possible refinement may involve budget prediction, trying to improve the average performance in cases that are not covered by the previous improvement. For instance, if all cores except one are idle, we will never reach the point where all cores consume their budget. This is a very tricky problem, because granting more

resources to the only running core may put at risk the tasks that may eventually spawn on the idle cores. The solution proposed in MemGuard is to predict the next memory usage from the previous history and, if a core is expected to use less budget than the allocated one, then redistribute its budget to the most demanding cores. If the prediction happens to be wrong, the core's budget is restored until the next period. This may put the system in a situation where the total budget of all cores is more than the minimum bandwidth, thus neglecting the per-core guarantees. The original implementation allows for this scenario, labeling it by the name of *underrun error*.
Over the many years, various works have tried to improve and extend the work presented in MemGuard. [4] introduced a distinction for cache refills and cache write-backs in the bandwidth computation: the intuition behind this is that writes are more costly in terms of DRAM transactions, therefore incurring more memory usage per write-back. In this case, the regulation is realized independently over the write bandwidth and read bandwidth. The issue is that we lack control over the aggregated contribution of reads and writes; on the contrary, we know that increasing the write bandwidth will surely have impacts on the read bandwidth due to bank interference. A further step forward is taken in [30], expressing the memory activity as a linear combination of reads and writes. This way, we can handle the case of interleaved reads and writes differently from the cases of read-only and write-only activity.
Many proposals tried instead to relocate the regulation algorithm to different processing elements. Decoupling the regulation logic and the regulated workload has the advantage to:

1. limit the regulation overhead on the regulated tasks, which permits to boost the frequency of the regulation loop;
2. reduce interference at the OS level (i.e., non-preemptive or higher priority handlers) which may cause fluctuations in the regulation loop.

MemPol [30] was the first to envisage this idea: the experiment involved the use of a heterogeneous platform comprising heterogeneous processing elements, all sharing the same addressing space. A cluster of Arm Cortex-A processors was running the workload to be regulated, while the control logic was running separately on a cluster of Cortex-R processors. This regulation was realized using the memory-mapped debug interface of the A processors, using the Performance Counters for profiling memory accesses and the Cross Trigger Interface for halting and resuming the cores.
A further step forward has been made in [10], where the regulation logic is instead realized in hardware in an FPGA. In the experiment, running on a CPU+FPGA board, the FPGA was regulating the traffic coming from a local Arm Cortex-A cluster. The FPGA on-chip monitors the bus transactions and halts and resumes the cores, using again the Cross Trigger Interface, but exploiting the internal trigger connections that can be signaled directly from the FPGA.

3.2.2 Memory bandwidth bounds

According to our model, main memory can be represented as a black box that provides a variable service rate (bandwidth) and distributes it in a non-deterministic way to the requesting CPUs. Therefore, to design any regulation algorithm, we need to know at least the worst-case guaranteed bandwidth. We will briefly review two major state-of-the-art solutions, that we will implement and analyze in Chapter 5 with a real scenario.

Inverse of the memory latency The simplest method for measuring a lower bound of the Guaranteed bandwidth is to measure the memory bandwidth of serialized accesses, that is, in other words, the inverse of the memory latency. For instance, this can be implemented as a pointer chasing over a linked list and doesn't require any particular detail about the hardware. Since serial accesses cannot exploit any memory level parallelism, we are sure that the measured bandwidth is lower than the Guaranteed one.
One issue of this method is that, in the measurement of the memory bandwidth, we are also considering the idle moments in between memory accesses, that is, the delay required to retrieve the data from memory, solve the dependency, initiate a new access and propagate to main memory.
The other issue is the difficulty in differentiating reads from writes: the pointer-chasing technique will only measure read bandwidth, as it only issues refills in cache. Any attempt to measure the write bandwidth will unavoidably cause both refills and write-backs in cache, especially in the case of write-allocate caches.
This approach was employed in MemGuard [27], where no differentiation between reads and writes was done.

Single bank performance If we are willing to trade portability for improving the accuracy of the measurements, we can opt for an approach like the one in [19]. If we know the mapping between address bits and DRAM banks, we can design a benchmark that performs accesses targeting only a specific bank. With this method, we can parallelize the accesses and, at the same time, completely neglect the effect of bank parallelism.

Differently from the previous case, we are not measuring latency, but bandwidth. This means that we can measure both the reads and writes in DRAM at the same time and extract separately the read and write bandwidth.

4. Regulating memory bandwidth from the JTAG port

By analyzing the previous results with a keen eye, we notice that the trend for the newer refinements has been to exploit specific hardware support to overcome MemGuard's shortcomings. For solutions like MemPol, we require a heterogeneous platform with application-specific cores alongside real-time cores, while solutions like MemCore require instead access to an FPGA on-chip.
We now ask ourselves: *can we offload the regulation algorithm away from the regulated cores, while making minimal assumptions on the underlying hardware?*
In our quest to answer such a question, we propose our JTAG-based memory bandwidth regulation approach, in an attempt to balance portability, accuracy and regulation frequency.

4.1. JTAG-based memory bandwidth regulation

Our work is again based on PMC regulation, but, whereas previous solutions leveraged memory-mapped interfaces, we envisage the usage of the external debug interface to access the debug registers, which has the advantage of:

- allowing control from the outside of the SoC, through the JTAG pins;
- mitigating contention at the bus level, as the debug components allow direct access to the CPU debug registers, without initiating transactions on the system bus.

The setup, therefore, requires two active elements:

1. the platform that should be regulated, exposing the JTAG pins;
2. the platform where the regulation control loop is running, that is able to drive the external JTAG signals (i.e., provide a GPIO or similar interface).

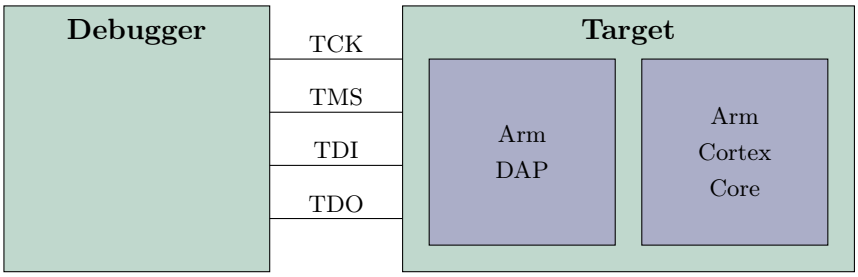


Figure 5: A high-level depiction of the regulation setup.

We further remark that, even though our solution employs the JTAG interface, it is possible to realize the same setup using either the SWD or SWJ interfaces with minimal changes to the source code. Since the JTAG is more widely supported, and since our evaluation platform did not support SWD, our implementation will be focused on the JTAG.
For simplicity, throughout the rest of this document, we will address the platform being regulated as *target*, and the platform where the regulation logic is running as *debugger*.
Once the connection between the target and debugger has been established, for instance, by connecting the respective GPIOs with jumper wires, we can start configuring the debugger with the target-specific specifications. Some vendors publicly disclose the information we require, but, since we base our approach on portability, we briefly outline a workflow for automatic recognition by the debugger. It will be of great value for platforms, such as ours, which completely lack any information about the debug components.

4.2. Connection initialization

To correctly access the debug registers of the target, we need to know:

- the number of TAPs connected,
- how the TAPs are interconnected (i.e., their topology),
- the length of the IR of each TAP,
- the availability of Memory Access Ports,

- the type of Memory Access Ports, that is, whether they are connected to the debug bus or system bus,
- the base address of each memory-mapped debug component.

Topology discovery As we have reminded in Chapter 1, multiple TAPs can be connected in series or parallel, in many possible configurations. The presence of multiple pins for the TMS signal or the TDI/TDO signals hints that some TAPs may be connected in parallel.
For each couple of TMS and TDI/TDO pair, we have a scan chain with many TAPs connected in series. To compute the number of TAPs in series, we can exploit a simple trick: the BYPASS instruction has a fixed encoding of all 1's regardless of the IR length and the associated DR register has a fixed length of 1 bit. To put all TAPs in BYPASS mode, we need to write enough 1's to fill all the IRs of all the TAPs. Since writing the IR means writing to a shift register, we can shift a sufficiently large amount of 1's on the external TDI and be almost sure that all the TAPs will be in BYPASS mode. A reasonable upper bound for this value is in the order of a hundred, as the usual length of an IR is between 4 and 8 bits, and it is very uncommon to have more than a few TAPs connected.
Once all the TAPs in the scan chain are in BYPASS mode, we can measure the number of TAPs by measuring the length of the scan chain. Arm implementations preload the BYPASS DR with a value of 0, making the process as simple as shifting 1's on the TDI and counting the number of 0's being shifted out of the TDO before reading again the 1's from the input.

IR length To compute the IR length of each TAP, we can exploit another design constraint of the JTAG standard: reading the IR will always return a read value of 1, regardless of the currently selected instruction. The binary encoding of the read value is characterized by a leading 1 with as many 0's as needed to fill the entire length of the IR. By scanning the IR chain, we can determine the length of each IR register in the series.

Memory Access Ports Each Access Port is selected by writing in the Debug Port SELECT register a specific 1-byte code. When an Access Port is selected, it must provide an Identification Register at a fixed location, which permits determining whether it is a Memory Access Port and which bus it is eventually connected to. Since the number of Access Ports that can be encoded with 1 byte is 256, we can simply iterate over all the Access Ports and scan all the APs.

ROM table To access the registers of the debug interface, 32-bit addresses are used. Each register address is computed by summing two parts: the base address of the debug component and the offset of the specific register within the component. The offset is fixed and constant among all implementations, where the base address of the device depends on the implementation. If the base addresses are not already known, it is possible to read them from a read-only memory on-chip. This is called the ROM table and stores in a table-like format all the memory-mapped debug components in the chip, their version and their base address in memory space. We will not cover the details of decoding the ROM table for retrieving the component's base addresses, as they are already well explained in the Arm manuals [1].
Once we have completely reverse-engineered the debug infrastructure, we can proceed with debug register manipulation: we can write in the IR scan chain to correctly pass instructions to the single TAPs, we can select the right Access Port and we know how the address space is organized. The building blocks we will use to realize memory bandwidth regulation are based on the following operations:

- configuring the PMU counters to count the desired events,
- manually modifying the PMC values,
- configuring the CTI to halt the cores on a PMC overflow,
- manually halting or resuming the CPU using the CTI.

Before implementing memory bandwidth regulation, we will more thoroughly analyze one missing aspect: the memory bandwidth bounds.

4.3. Memory regulation

Using our JTAG-based approach, we implemented two regulation algorithms:

- the first is directly taken from literature, as we readapted the original MemGuard [27] algorithm to operate through the JTAG interface;
- in addition to that, we devise our own regulation algorithm, that we call *MemAdapt*, specifically designed to complement JTAG-based regulation.

4.3.1 MemGuard/JTAG model

The difficulties encountered in implementing MemGuard lie in the fact that, while MemGuard could associate software handlers to PMC overflow interrupts, our approach does not allow the debugger to promptly run software on each overflow: it should periodically poll the CPU, checking for any PMC overflow.
Whereas it is totally feasible to implement such a scheme, we prefer, instead, to entirely devote the debugger resources to making the regulation period shorter, rather than increasing the regulation period and polling the PMCs in between.
As a result, our implementation will not resume the CPUs when they have all exceeded their budget. We also refrain from implementing the period adaptation scheme, as it will hinder real-time guarantees, and prefer, instead, to run the regulation loop at the maximum attainable frequency.
However, we decided to extend MemGuard with the improvement in [4], using both the cache refills and cache write-backs counters. We program the PMU to halt the CPU when the first of the two counters overflows, then we restore both counters and resume the CPU.
The final implementation is quite simple; it requires just a few writes in the debug registers and thus can run at a very high frequency.

4.3.2 MemAdapt model

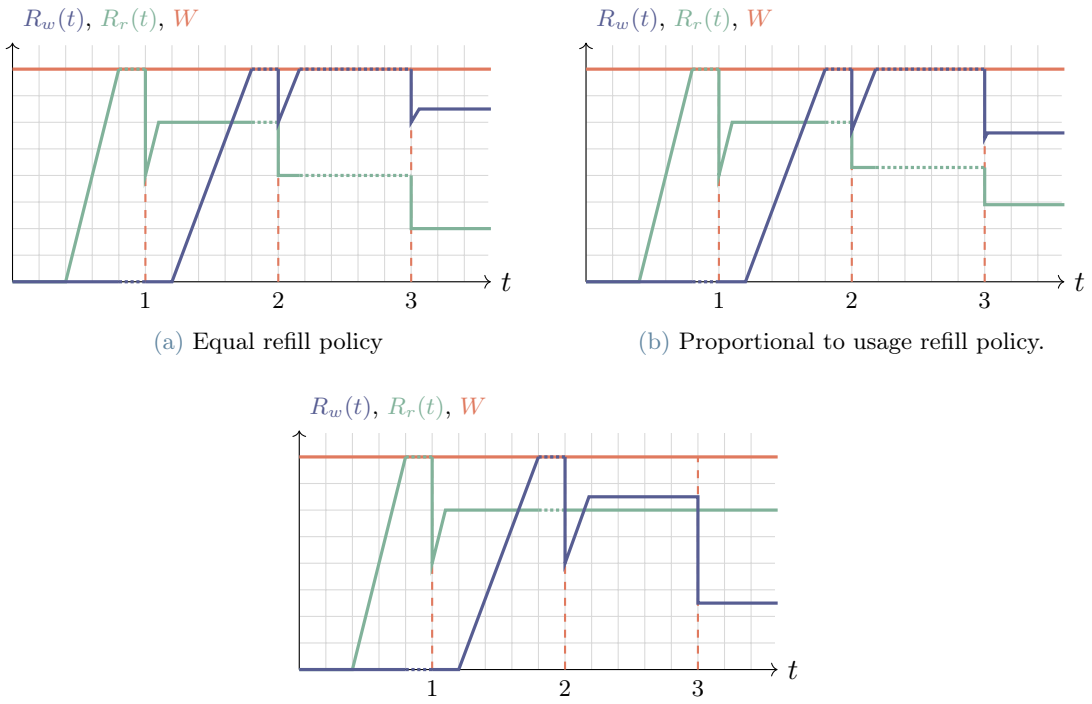
MemAdapt was created on our experience from the previous works and from the limitations of our JTAG approach, while at the same time focusing on adaptability, hence the name *MemAdapt*. Our design choices can be summarized as follows:

- we prefer to use the widely available Debug components functionalities of the PMU and CTI for halting the cores, since PMC polling only works at very fast frequencies;
- we compute the maximum bandwidth budget as a linear combination of reads and writes to provide adaptability to any architecture;
- we consider the introduction of the regulation window to be a better approach than simply increasing the regulation period, as it grants flexibility to the implementation without hindering guarantees;
- for improving the average case without altering the worst case, we adopt a technique similar to MemGuard, which allows free memory usage when all CPUs consume their allocated budget.

We use two counters to count the LLC refills and write-backs, and halt the CPU when the first of the two overflows. The initial counters' values are set in such a way to overflow after a fixed number of events, which is configurable and different for refills and write-backs. When an overflow occurs, we don't reset the counter to the starting values, but, instead, we subtract (using modular 32-bit algebra) a constant from each of them. This constant represents the budget per period and is computed starting from two parameters, β_w and β_r, to match the refill amount per period for reads and writes. Avoiding resetting the counters also brings another benefit: we are not neglecting the overshooting due to the latency of halting. As we know, debug entry is taken precisely, so pending memory operations are not suspended and will contribute to the PMC value.
When the CPU is idle, the counters decrease on each periodic refill, so that, when a new task spawns on the CPU, more events can be submitted before the core will be halted. This effectively allows building up unused budget over many periods. Since infinite reclaiming may neglect real-time guarantees, we impose a minimum bound on the PMC counted value, such that the maximum usable budget matches the budget of a chosen number of periods, which we call the sliding window size W.
Special care should be taken when designing the refill policy of the counters: we choose β_w to limit write-backs to β_w per period in write-only performance, and we choose β_r to limit refills to β_r per period in read-only performance. In case of mixed usage of reads and writes, we need to properly distribute the available budget to each PMC. To accomplish this, we have explored 3 different approaches.

Single refill The simplest approach is to refill only one counter at a time, that is the one that is closest to an overflow, or, if the CPU is halted, the counter that caused the overflow. This effectively guarantees that each counter cannot receive more budget than the allocated refill per period, and correctly distributes the budget between refills and write-backs based on the current needs. This choice is better suited when we want to optimize burst accesses of the same type, as we periodically refill only the counter we are using.

Equal refill Another straightforward approach is to equally refill both counters by half of the respective budgets. Even when one counter overflows, we still restore budget to both of them, hoping that the following memory accesses will cause both refills and write-backs in the LLC.



(a) Equal refill policy

(b) Proportional to usage refill policy.

(c) Refill lowest budget.

Figure 6: A graphical representation of the different proposed refill policies. The red line represents the maximum budget that can be accumulated, whereas the green and blue lines represent the used budget from reads and writes, normalized by their refill factor ($R_i(t) = [\text{used budget for } i]\,/\beta_i$). The lines are dotted when the CPU is halted.

Proportional refill As a trade-off between the previous methods, we can employ a "smarter" refill policy that restores the budget proportionally to prior usage. This is more flexible, but the cost for flexibility is the complexity of the implementation: to avoid underflows and overflows in the 32-bit operations, several constraints on the algorithm parameters had to be set. This approach, as well, is able to adapt refills to any access pattern. All the previous cases may waste precious budget when the counter to refill has accumulated the maximum allowed budget. In this situation, we implement a budget-stealing technique that converts the budget that would exceed the maximum threshold into usable budget for the other counter. For instance, if the CPU is performing only refills, all 3 algorithms will now perform the same, providing β_r refills each period; this is because, sooner or later, the write-back counter will saturate and the exceeding budget will be donated to the refills counter.

4.3.3 Memory budget allocation

Before each algorithm can enforce the target bandwidth, we first need to convert it to an amount of budget, for both MemGuard and MemAdapt. We will denote as BW_B^r and BW_B^w the assigned read-only and write-only bandwidth .

MemGuard When assigning the MemGuard budget, we must be very careful as the algorithm does not regulate the linear combination of read and write bandwidth, but instead put a hard limit on the write bandwidth and, independently, on the read bandwidth. Therefore, we should choose the memory bounds such that the combined read and write bandwidth will always be within the linear combination of BW_B^r and BW_B^w. Geometrically, this means that if we plot the MemGuard budget as a point (Q_w, Q_r), it should lie on the line from $(0, BW_B^r)$ to $(BW_B^w, 0)$, or, in formulas:

$$\frac{Q_w}{BW_B^w} + \frac{Q_r}{BW_B^r} = 1.$$

Since any choice satisfying the above relation is correct, our implementation will pick the budget parameters that maximize the product $Q_w \cdot Q_r$, corresponding to the assignment:

$$Q_w = \frac{1}{2} BW_B^w, \quad Q_r = \frac{1}{2} BW_B^r.$$

MemAdapt Since our regulation algorithm does not need to perform complex operations with the PMC values, we can perform a more straightforward assignment:

$$\beta_w = BW_B^w, \quad \beta_r = BW_B^r.$$

Our algorithm also allows us to configure how the budget is distributed to the two counters by choosing one of the proposed refill policies. To avoid errors due to the integer arithmetic, we prefer to use the Equal Refill policy, which is better tailored for 32-bit integers.

5. Evaluation

We have successfully implemented and evaluated our JTAG-based regulation scheme on a real platform, using a Nucleo STM32F429 as debugger and a Raspberry Pi 4 Model B as target. The choice of this pair is motivated by the high availability of such hardware and because the respective GPIOs are both powered by 3.3 V voltage. As this is not strictly mandatory, having equal voltage allows us to just connect the two boards with simple jumper wires, without the need to bridge the voltage via other components in the middle.
The Raspberry Pi uses the Broadcom BCM2711 chip [5], which contains a quad-core Arm A72 cluster, that is based on the *Arm v8.0 A profile* architecture, and features a 1 channel 4 GB LPDDR4 DRAM memory. The cache hierarchy is made up of 2 layers of caches: a private L1 cache, in Harvard configuration with 48 kB data cache and 32 kB instruction cache, and a shared 1 MB L2 cache. More details about the memory hierarchy are listed in Table 7.

Cache	
Cache line size	64B
L1D total size	48kB in 3 ways
L1I total size	32kB in 2 ways
L2 total size	1MB in 16 ways
L2 bank bits	4,5,6 (8 banks)

DRAM	
Total size	4GB
Channels	1
Ranks	1
Banks	8
Bank bits	12,13,14

Figure 7: Additional details about the Raspberry Pi 4 Model B (4GB) cache and main memory organization.

It exposes a 40-pin GPIO interface that comprises the standard JTAG pins bundle. Every component in between the external pins and the CPU registers is undocumented, making the reverse-engineering techniques presented in the previous chapter necessary.
It is running Raspberry Pi OS, with the Raspberry Pi Foundation's Linux Kernel fork. We used three different Kernels that we compiled for this purpose:

- Linux Kernel 5.15.92 patched with `PREEMPT_RT` and Huge Pages enabled, to run the benchmarks that required Huge Pages;
- Linux Kernel 5.15.92 patched with `PREEMPT_RT` and PALLOC [28], to run the experiments that required cache partitioning;
- Linux Kernel 5.15.92 patched with PALLOC, to run the original MemGuard with cache partitioning;

In all cases we added the kernel command line option `isolcpus=3` to disable automatic scheduling on CPU core 3, so that we schedule benchmarks on this core and minimize the OS interference.
The Nucleo STM32F429ZI [20] features a single Arm Cortex M4 core with 2 MB flash memory, operating at the frequency of up to 180 MHz. It exposes a highly configurable GPIO interface, that can work at frequencies between 100 MHz and 180 MHz [21] if powered with a stable voltage, with a load capacitance in the 10-30 pF range.
The connection between the Nucleo board (debugger) and the Raspberry Pi (target) is realized using jumper wires connecting the TCK, TMS, TDI, TDO, TRST and GND pins of the Raspberry Pi to arbitrary GPIO pins of the STM32 board. Each pin-to-pin connection uses two 24 AWG copper wires in series, each being 20 cm long. A wiring layout is shown in Figure 8.
We configure the debugger's GPIO to work at maximum speed and disable all pull-up and pull-down resistors for the pins. As it may seem beneficial to use the pull-up resistor to assert the TRST pin high, our experiments proved that the pull-up resistor may cause fluctuations in the TRST signal on the target JTAG circuit, making

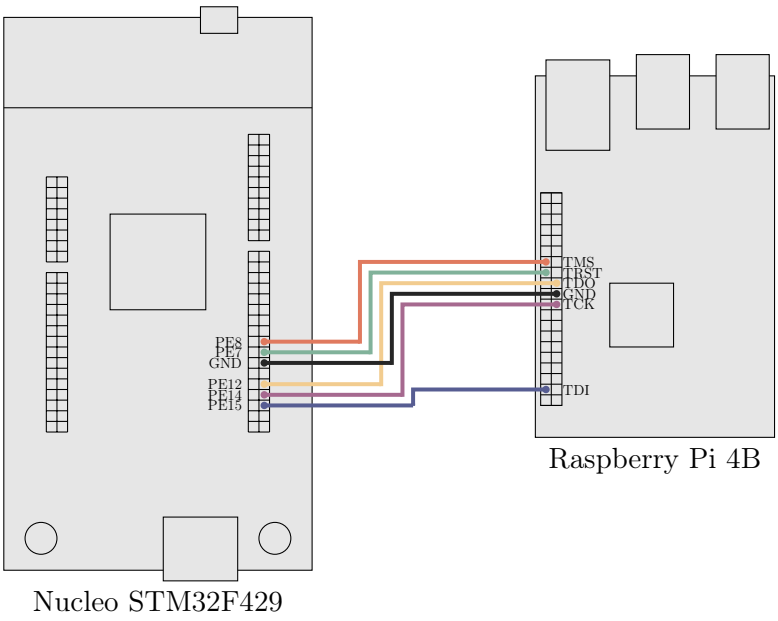


Figure 8: The wiring layout between the debugger (left) and the target (right).

the circuit state inconsistent. With the exception of the TDO pin, which we configure as input[3], all the other pins are configured as digital output and always written from software, using the memory mapped GPIO interface. Since the JTAG circuit is asynchronous, we don't need any timer/clock for now, and the TCK pin state is changed as soon as needed, as fast as the board allows it. From the Raspberry Pi side, we configure the pins in Alternate Function mode to enable access to the JTAG circuit.

5.1. Probing the debug interface

Before we can evaluate the maximum access speed to the debug components, we need a way to assess the correctness of each access, which will be our first goal. For now, we keep the board frequency at the default value of 72 MHz and add long delays between each pin transition. Since the debug details for the Raspberry Pi board are not disclosed by the producer, we start with the reverse-engineering framework we discussed in Chapter 4.

DR scan chain with BYPASS	`0b...11111110`
IR scan chain	`0b...11110001`
IDcode	`0x4ba00477`
AP0 IDR	`0x24770002`
AP1-127 IDR	`Not implemented`
Core n Debug base address	`0x80410000` + `0x100000`$\times n$
Core n CTI base address	`0x80420000` + `0x100000`$\times n$
Core n PMU base address	`0x80430000` + `0x100000`$\times n$

Table 1: The reverse-engineered debug information for the Raspberry Pi 4 Model B.

The readings, shown in Table 1, report that the external debug access port is made up of only one TAP, whose IR length is 4 bits. There is only one Access Port, which is also a Memory Access Port connected to an APB debug bus. Our CPU only implements a minimal set of debug features; thus, the number of components listed in the ROM table is limited to 3 different types: PMU, CTI and general Cortex Debug registers.

[3]For input/output directions, notation are generally considered with respect to the target. That means that the Test Data Output (TDO) is an output for the target, thus an input for the debugger.

Regarding the PMU event counters, our platform can only count a limited set of events, ruling out any DRAM-based event counters. To count reads and writes in DRAM, we resort to the `L2D_CACHE_REFILL` and `L2D_CACHE_WB` events. Respectively, they count the number of times data is fetched or written into any memory hierarchy lower than the L2, which, in our case, only leaves the main memory.

5.2. Access speed

We can now perform an evaluation of the maximum achievable access speed. We execute a fixed number of accesses, we read the IDCODE register and compare the read result with the known value. Then, we measure the time elapsed using the STM board timers and compute the error rate, that is, the observed single-bit errors (i.e., Hamming distance) between each read value and the real IDCODE.
The most common error we observe is that the value we read appears to be shifted by 1 bit with respect to the real one. We attribute this issue to the delay from the moment we instruct a change on the JTAG finite-state machine to the moment the actual bit is asserted on the transmission line. That means we are reading the pin too soon.
We repeat the test:

- by changing the board frequency and
- adding a variable delay before reading the TDO pin.

We increase the board frequency starting from the base value of 72 MHz, up to 160 MHz. We don't reach 180 MHz, as our experiments showed instabilities as we approached that frequency, like, for instance, within the USART, making it impossible to retrieve the measurements.
The delay before reading the TDO pin is realized in software by inserting NOP instructions before the read. We also notice that the delay caused by stalling the CPU is proportional to the clock frequency. As we can see in Table 2, the maximum feasible JTAG access frequency can be obtained at 85 MHz with a 2 NOPs delay and at 129 MHz with a 3 NOPs delay. After closer inspection, we can observe that in both cases the delay is $2/85\,\text{MHz} = 23.53\,\text{ns} \simeq 3/129\,\text{MHz} = 23.26\,\text{ns}$.

\Freq Delay	**72**MHz	...	**85**MHz	**86**MHz	**87**MHz	**88**MHz	**89**MHz	**90**MHz	**91**MHz	...
2 NOPs	155.6kHz ±0.6kHz	...	184.3kHz ±0.9kHz	0.004%	18.73%	18.75%	21.79%	37.50%	37.50%	...

\Freq Delay	**72**MHz	...	**129**MHz	**130**MHz	**131**MHz	**132**MHz	**133**MHz	**134**MHz	**135**MHz	...
3 NOPs	145.5kHz ±0.6kHz	...	261.5kHz ±1.8kHz	0.409%	18.75%	18.75%	18.75%	34.23%	37.50%	...

\Freq Delay	**72**MHz	...	...	...	...	...	...	...	...	**160**MHz
4 NOPs	136.5kHz ±0.5kHz	...	...	...	...	...	...	...	...	305.4kHz ±2.4kHz

Table 2: A summary of the maximum achievable access frequency. The columns refer to the board clock frequency, while the rows refer to the delay introduced, in terms of number of stall instructions issued before reading the TDO pin. If the detected error rate is 0, the achieved access frequency (as 32 bits words per second) is shown (green cells) with the respective accuracy, which depends on the timer precision. When the error rate is greater than 0 (red cells), we report the observed error rate with respect to single-bit errors.

In all tests, we observed a maximum error rate of 37.5%, which can be written as 12/32. In fact, 12 is the Hamming distance between the 32 bits `IDCODE` and the `IDCODE` shifted by 1 bit.
For the rest of the experiments, we will fix the clock frequency at 160 MHz and use a 4 NOPs delay, which allows us to read a register at the frequency of 305.4 kHz.
To read the CPU registers, we need to perform multiple accesses: to configure the debug port, to write the address to read from, then perform the access and, eventually, check for errors. With the chosen clock frequency, we measured the time required to read a debug register, for example, the Component ID register 0, for each CPU, and then check for errors.
Leaving out the initial operations required to configure the debug port, we need to issue 4 read operations in the Access Port, 1 read and 1 write in the Debug Port. If we implement the AP accesses using the chaining scheme presented in Chapter 2, the total number of operations becomes $(4 \cdot 2 + 1) + 1 + 1 = 11$ DR operations. The measured value is 49.2 µs, corresponding to 20.3 kHz, and is in line with our assumptions.

5.3. Benchmarks

Through the rest of this chapter, we will extensively use different benchmarks, some synthetic, some not. We will not delve deep into the implementation details for each one, preferring instead to motivate our choice with respect to our needs.

IsolBench IsolBench was first presented in [24] and, since then, has been extensively used by successive works in literature. It is therefore a well-established and reliable benchmark suite, that allows us to compare our work with analogous ones in literature.
The `bandwidth` benchmark performs sequential accesses over an address space that is allocated by either calling `malloc` or by mapping a Huge Page. It calculates the average memory bandwidth measured as the ratio between the number of accesses and the benchmark duration. This benchmark is a good choice for extracting the maximum achievable bandwidth, as most of the accesses are to sequential memory locations, thus exploiting hardware optimizations.
The other useful benchmark is `latency`, which simply implements a linked list pointer chasing. It returns the average latency by dividing the benchmark duration by the total number of memory accesses.

San Diego VBS The San Diego Vision Benchmark Suite [25] is our candidate for the analysis of a real-world workload scenario. It is a bundle of diverse vision applications, coming with different input formats, each affecting the benchmark working set size. Image manipulation is known to be a memory-intensive task, especially if the image does not fit in cache.
Since the original version is designed to work only on x86 architectures, we use the version that is part of the RT-Bench benchmark suite [13], as it is compatible with Arm architectures.

LL-Mem-Stress LL-Mem-Stress is a synthetic benchmark that we designed to create a configurable stressor based on linked list pointer chasing.
Differently from the IsolBench `latency` benchmark, which only allows random or sequential accesses, we allow the user to configure the access pattern from a configuration file. This is possible because when mapping a Huge Page, we can be sure that the bits that encode the offset within the page have a 1-to-1 mapping between the virtual and their respective physical address. We also allow a configurable level of parallelism, by mapping multiple Huge Pages and traversing multiple linked lists at the same time.
The configuration requires a list of bits to select the bits that should change over multiple iterations and their order. To compute the next address, we flip the first bit in the list and, if the bit transitions from 1 to 0, we say that we have a carry. If we have a carry, we also flip the next bit in the list, check again for carry, and continue recursively, until: we have no carry, or we have visited the full list. With the order obtained, we prepare the memory inside a Huge Page and build a linked list that traverses all the chosen addresses.
For example, choosing a list of `[0,1]` will induce an order of $0 \to 1 \to 2 \to 3$, while the list `[1,0]` will induce an order of $0 \to 2 \to 1 \to 3$.

5.4. Choosing the regulation budget

Before starting the actual memory regulation, we only need the last building block: the estimation of the guaranteed bandwidth, which we need for assigning a memory budget to each core. For this purpose, we will now evaluate and compare the methods presented in the previous chapter.

5.4.1 The guaranteed memory bandwidth

The first method required the latency of a single memory access. For this purpose, the IsolBench `latency` benchmark exactly matches this purpose, and we simply report the measured value in Table 9.
In order to proceed with the second method, we need a benchmark that can perform memory accesses targeting the same bank. Unfortunately, USTRESS, which was used in [19], is not publicly available, while IsolBench cannot fine-tune access to a specific bank. This is the reason that led us to design our own benchmark, LL-Mem-Stress, that can be adapted for this use case.
We configure the benchmark to access all the memory locations within a Huge Page that map to memory bank 0, and then compile two versions: one that only performs reads by iterating over the linked list, and the other that also performs a write at each node of the list. We then execute each of the two while periodically polling the PMU counters from the debugger.
With a fixed polling period of 50 ms, we report in Table 9 the average read and write bandwidth of each scenario, converted into MB/s.

As we can see, it was not possible to compute a standalone read or write bandwidth. The effect is particularly visible in the write-intensive scenario because in a write-back cache every write to memory is triggered by a cache miss in the LLC.
For the only purpose of computing the regulation budget we can consider a purely virtual write-only and read-only bandwidth, given by the linear extension of the 2 points from the previous experiment. These values, reported in Table 9, have only an abstract meaning, but greatly simplify the memory budget calculation, as we can reuse the formulas from Section 4.3.3.

Algorithm	Read	Write
Serialized access bandwidth	427.45 MB/s	-
Single bank reader bandwidth	2846.80 MB/s	0.213 MB/s
Single bank writer bandwidth	755.45 MB/s	631.95 MB/s
Single bank read-only (virtual) bandwidth	2847.50 MB/s	-
Single bank write-only (virtual) bandwidth	-	860.15 MB/s

Figure 9: Guaranteed bandwidth estimations.

5.5. Evaluation of memory bandwidth regulation

We have now satisfied all the requirements to correctly implement the presented algorithms using our JTAG-based approach. We can proceed to prove the efficacy of this method in a real scenario with our experimental setup. In the following tests, we will measure:

- the effectiveness of memory regulation, as the capability to enforce the chosen memory budget;
- the slowdown induced by regulation.
- how a real-world workload performs in the presence of memory interference;

For all cases, we will fix the BW_B^r and BW_B^w regulation parameters starting from the Guaranteed memory as single bank bandwidth, using the computed write-only and read-only bandwidth. We assigning each CPU a fraction equal to 20% of this value, resulting in:

$$BW_B^w = 172.03\,\text{MB/s}, \quad BW_B^r = 569.50\,\text{MB/s}.$$

Additionally, MemAdapt has been tuned with a sliding window of 4, and we set the refill policy as Refill Equal. Alongside MemGuard/JTAG and MemAdapt, our experiments will also feature the authentic MemGuard implementation, using a kernel module. We have chosen the MemGuard implementation that matches our MemGuard/JTAG, that is the one regulating both the read and write bandwidth, and no optimizations were enabled. The budget allocation rules follow the same for MemGuard/JTAG described in Section 4.3.3.

5.5.1 Worst-case memory bandwidth under regulation

We can evaluate the effectiveness of each regulation framework by assessing their ability to enforce the chosen memory budget. When the actual memory usage stays below the allocated budget, resources are under-utilized; conversely, if applications exceed their assigned budget, the system becomes unsafe and memory isolation is compromised.
To perform this experiment memory regulation was combined with periodic polling of the performance counters from the JTAG interface, comparing the memory bandwidth parameter of each regulation framework with the counters readings. We use the IsolBench benchmark, spawning one instance on the isolated CPU core 3, scheduled with `SCHED_FIFO` with priority 10, and then capture 4000 sequential samples from the JTAG. The same period is used for sampling the counters and for memory regulation, of 300 µs.
The experiment is repeated for each regulation framework and for each access pattern of IsolBench (read-intensive and write-intensive). Then, starting from the number of LLC misses, we compute the maximum observed memory bandwidth over different interval lengths, starting from a single period (300 µs) up to 60 periods (18 ms).
For comparing different algorithms, we plot in Figure 10 the budget usage as a scalar value, using the read-only and write-only bandwidth regulation parameters; a value of 1 means that the consumed budget matches the allocated one. We used the following formulas, where BW_B^w and BW_B^r are the read-only and write-only parameters (the same as in Section 4.3.3), and BW^w and BW^r are the read and write bandwidth measured from the samples.

$$U_{memguard}\left(BW^w, BW^r\right) = \max\left(\frac{BW^w}{BW_B^w/2}, \frac{BW^r}{BW_B^r/2}\right), \quad U_{memadapt}\left(BW^w, BW^r\right) = \frac{BW^w}{BW_B^w} + \frac{BW^r}{BW_B^r}.$$

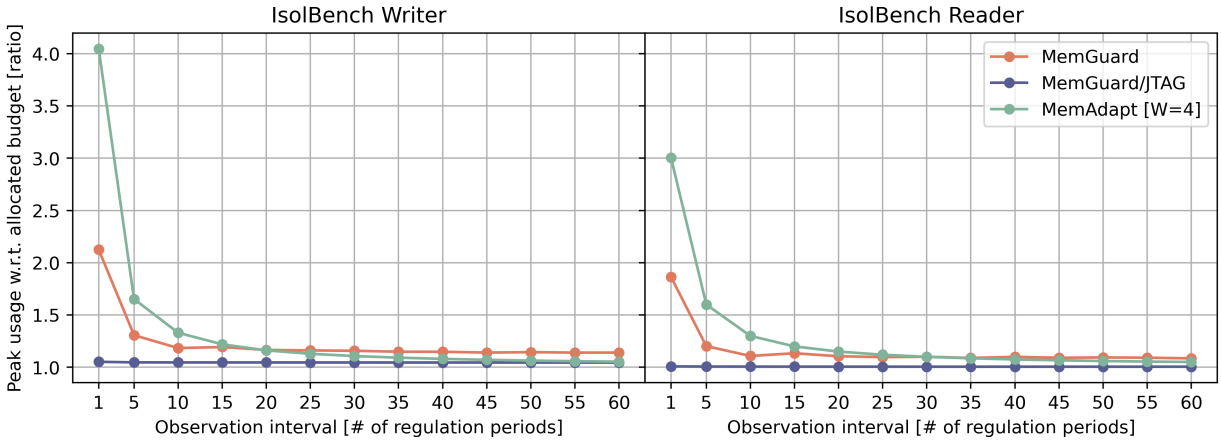


Figure 10: Consumed memory budget under regulation. On the vertical axis the budget usage as the ratio of the maximum consumed budget with respect to the allocated budget. If this value is greater than 1 the allocation is unsafe, if it is smaller than 1 it denotes under-utilization, while a value of 1 is the ideal one. The left panel shows the results for an IsolBench write instance, while the right panel shows the same test for IsolBench read.

As expected, the original MemGuard implementation shows a considerable difference between the imposed budget limit and the measured used budget. Over the duration of a single regulation period, the peak budget usage reached a value of about 2 (2.12 IsolBench writer, 1.86 IsolBench reader), that we attribute to the misalignment of the regulation period and the polling loop. Different is the result for longer intervals, as the maximum observed bandwidth never converges to the allocated budget: after 60 regulation periods the IsolBench writer exceeded the budget by a factor of 1.14 and the IsolBench reader by a factor of 1.09. The reasons behind this are:

- the delay in the halting logic, that in MemGuard is the time required to pass execution to the MemGuard idle task,
- and regulation algorithm design, as the excess memory operations are never taken into account.

Regulation through JTAG and debug registers achieved instead more consistent results: after 60 ms the used budget was exceeded by a factor up to 1.04 for MemGuard/JTAG and 1.05 for MemAdapt. For MemGuard/JTAG the explanation is the same as the previous case, with the difference being that the slowdown is caused by delay for debug halting; for MemAdapt the difference is due to the delay that elapses from the moment a PMC is read to the moment the new value is written back to it. MemAdapt showed a peak for an interval of one regulation period (4.04 IsolBench writer, 3.00 IsolBench reader), that is inline with expectations, as the sliding window parameter was set to 4.

5.5.2 Slowdown due to regulation

In this section, we want to compare the overhead of the different regulation algorithms on the system performance. We remind that the slowdown suffered by applications is determined by multiple factors:

- the regulation overhead,
- the allocated memory budget, and
- the access pattern of the current application.

We use the San Diego Vision Benchmark Suite to simulate a real workload; we run all the benchmarks in the suite and choose `cif` as input size, which is the largest input that resulted in feasible execution times while still keeping most of the tasks memory-bound. For creating background memory interference, we spawned IsolBench instances on all the other cores.

We run 10 instances of each benchmark, scheduled with `SCHED_FIFO` with a priority of 10 on core 3, at a constant CPU frequency of 1.5 GHz, and repeat the test:

- for each benchmark in the SD-VBS;
- without any regulation and for different combinations of regulation schemes and parameters;
- with no interference, with read-intensive interference and with write-intensive interference.

The different regulation choices include: original MemGuard with regulation periods of 50 µs, 150 µs and 300 µs, MemGuard/JTAG with regulation periods of 300 µs and 150 µs, and MemAdapt with a regulation period of 300 µs and sliding window of 4.
We compute the average slowdown due to regulation as the average execution time under regulation divided by the average execution time of the unregulated run. We report in Figure 11 the obtained results, classified by the benchmark instance, used regulation framework and interference pattern.

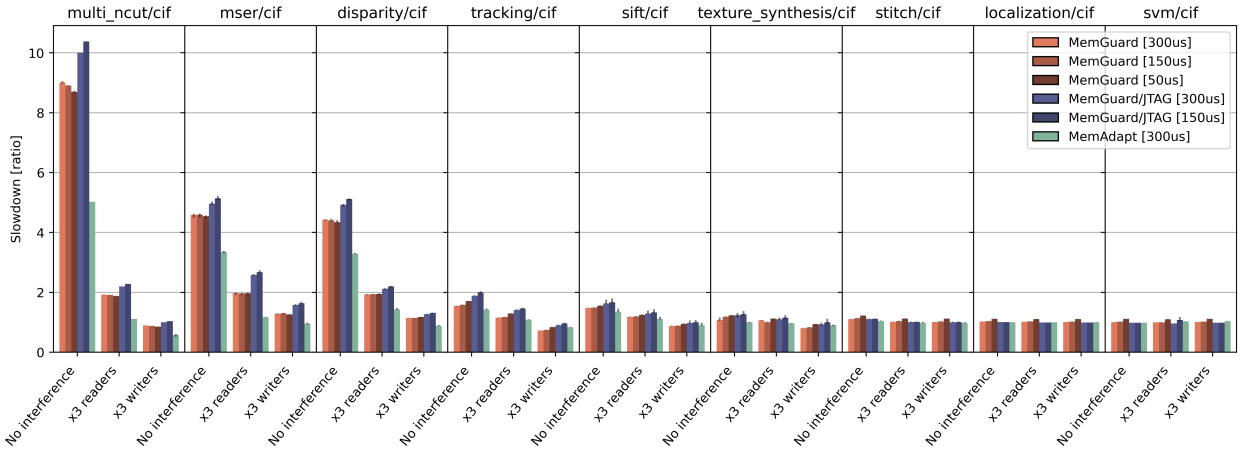
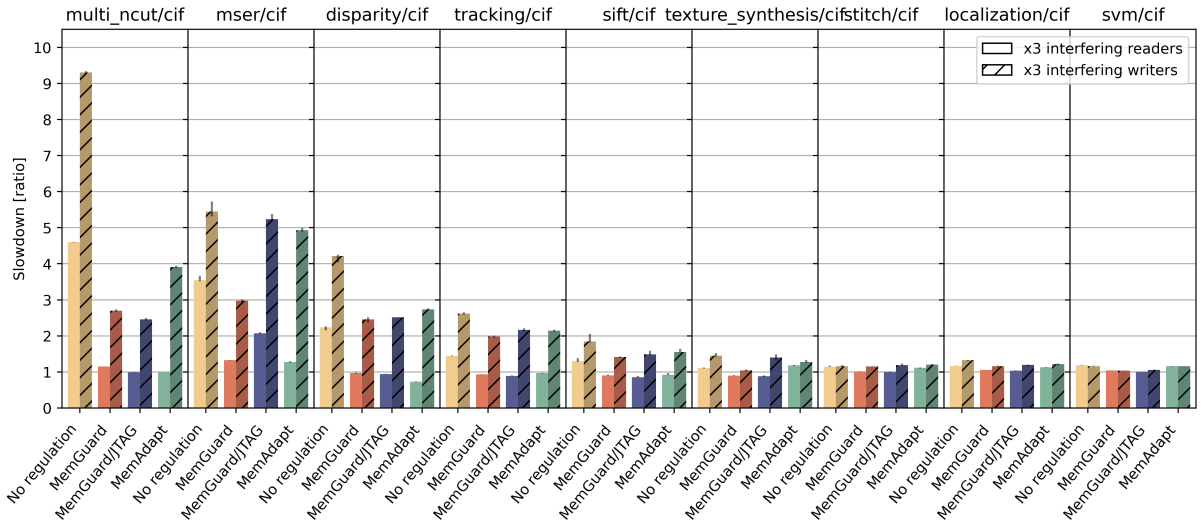


Figure 11: Overhead due to regulation for different regulation frameworks and interference patterns. On the vertical axis the average slowdown due to regulation, as ratio of regulated over unregulated run, under the same interference conditions; on the horizontal axis the regulation frameworks (color of the bars) and the interference conditions. The bars represent the average computed value, while the vertical gray lines show the minimum to maximum observed value.

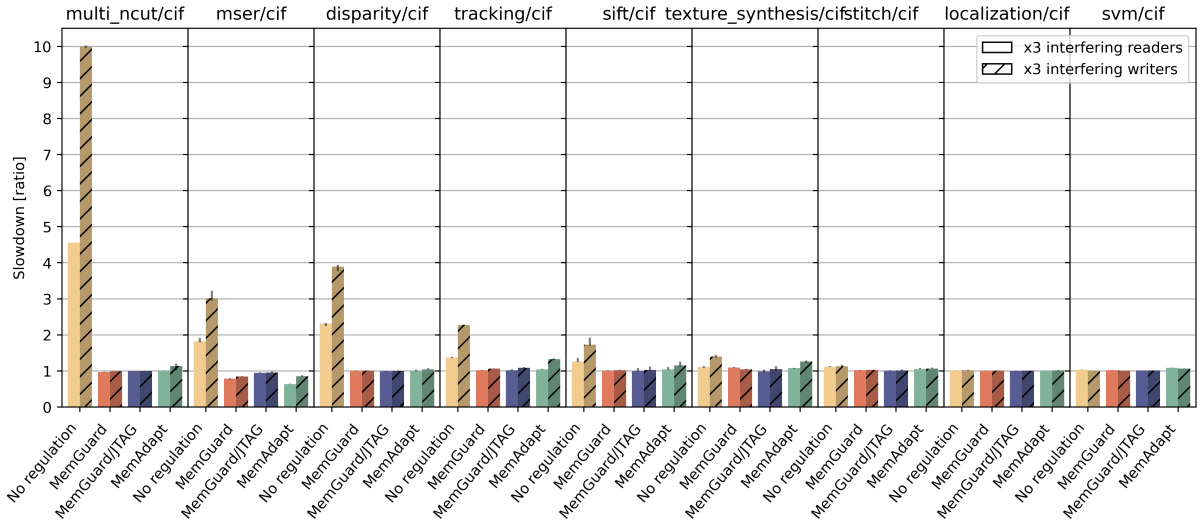
We can make a major distinction between memory-bound (`multi_ncut`, `mser`, `disparity`, `tracking`, on the left) and cache-bound (`stitch`, `localization`, `svm`, on the right) tasks: the first class is mostly affected by budget allocation, while the latter by the regulation overheads. MemGuard/JTAG caused a higher slowdown with respect to original MemGuard in the case of memory-bound tasks: this is a result of MemGuard's inaccurate budget allocation, as seen in Section 5.5.1. On the other hand, cache and CPU-bound tasks showed the opposite trend, putting into highlight MemGuard's software overhead, which is especially visible in the 50 µs regulation period case. Overall, MemAdapt showed better performances than both versions of MemGuard, achieving a more efficient memory regulation in all scenarios.

5.5.3 Real-world workload analysis

Our real-world workload is again the San Diego Vision Benchmark Suite with IsolBench tasks causing memory interference. We run all the benchmarks with the same conditions as Section 5.5.2, but this time we measure the slowdown due to interference, and not regulation. Mathematically, this means that, for each benchmark and for each type of regulation, we compute the slowdown as the benchmark execution time affected by interference divided by the execution time without interference. When memory interference is correctly mitigated, this value should be close to 1.
As we can see in Figure 12a, that is not the case, as we mitigated DRAM interference, but allowed contention at cache level. However, from the picture, we can still appreciate a reduction in the slowdown when either of the 3 algorithms is used.
Just to be completely sure about our results, we repeated the same benchmark with cache partitioning, to mitigate the effects of mutual cache line evictions. Even though we still have other sources of interference, such as the cache bank contention, the slowdown incurred is negligible with respect to the main memory interference. We realized cache partitioning using page coloring, with the PALLOC kernel patch proposed in [28]. We pay careful attention to the choice of the PALLOC bit mask, as we may unintentionally partition DRAM banks, too. We decide to split the cache across bit 15, which is not used to derive the DRAM bank, nor to compute the L1 cache set. We assign one partition for the foreground IsolBench task, and the other partition for all the background interfering tasks.
The results in Figure 12b show that interference has now a minimal impact on the execution times for all the regulation algorithms, proving that our regulation framework was correctly implemented and apt for the purpose.



(a) The effect of interference on the San Diego VBS without cache partitioning.



(b) The effect of interference on the San Diego VBS when cache partitioning is used.

Figure 12: The effect of interference on the San Diego VBS, under different memory regulation frameworks. The vertical axis shows the slowdown suffered by the foreground application, when interference (IsolBench) is spawned on neighboring cores, under the same type of regulation. The colors identify the regulation, while the hatches on the bars differentiate the interference type (read-intensive without hatches, write-intensive with hatches). The top plot (a) shows the basic experiment, while the bottom (b) repeats the same with cache partitioning.

5.6. Comparison of the regulation algorithms

All the proposed regulation algorithms succeeded in the task of protecting the CPU from memory interference caused by co-running applications. Each of those algorithms accomplished the same task in different ways, exposing different strengths and weaknesses. We briefly perform a comparison of such algorithms, and how they compare with MemGuard, the state-of-the-art implementation.

MemGuard/JTAG MemGuard/JTAG achieved very positive performance benefits; its simplicity allowed us to reach the shortest regulation period for the JTAG regulation, reaching a minimum of 150 µs.
At the same time, with respect to the original MemGuard implementation, we achieved a more coherent budget allocation, reducing the average error between the assigned and measure budget from 14% to 4%.
Hosting the regulation loop separately from the OS software achieved a neater and more precise regulation: when the original MemGuard was executing we observed numerous warnings due to the periodic handler overruns and due to kernel threads starvation. Our MemGuard/JTAG implementation completely removed OS dependencies and software interference, resulting in a cleaner and error-free Linux experience.

MemAdapt Our MemAdapt preserves the same isolation properties of MemGuard/JTAG, while granting more flexibility in the algorithm parameters and more accurate bandwidth enforcement.
MemAdapt achived the best regulation overhead over regulation effectiveness ratio, out-performing both MemGuard and MemGuard/JTAG in all experiments. This is enabled by the algorithm choice to limit linear combination of reads and writes, rather than doing it separately. We remind that MemGuard required to limit the read-only and write-only bandwidths to take into account the read-and-write-heavy scenario: given that our platform employs a write-back cache, we can say that we optimized the write-intensive (which causes both cache refills and write-backs), but severely limited read-intensive applications (that only causes refills). The effect on San Diego Visual Benchmark Suite further confirmed these limitations.

6. Conclusions and future developments

This thesis demonstrated that JTAG-based memory bandwidth regulation is a viable, portable, and effective solution for mitigating memory interference in multi-core systems. As demonstrated in Chapters 4, this approach enables external control of memory resources without any hardware or software modification, while keeping the platform requirements at a minimum.
By introducing MemAdapt, we achieved a balance between safety, efficiency, and adaptability, offering a valid alternative to state-of-the-art implementations.
The portability of our solution has been proved on the field, by adopting a COTS platform (Raspberry Pi 4B) that provided minimal architectural information and correctly implemented MemAdapt and one state-of-the-art memory regulation algorithm: MemGuard.
In Chapter 5, we have conducted a thorough test campaign, confirming that our goal of providing memory isolation to co-running applications has been finally reached.

6.1. Contributions

Throughout this thesis, we have made several contributions to advance the state of the art, both in the context of resource management and in the context of memory bandwidth regulation. The most significant contributions are:

1. We have re-imagined the debug infrastructure to perform an active orchestration on the target's memory resources, opening the possibility of moving the control logic away from the controlled instance and establishing the control loop using the widely available JTAG interface.
2. We devise a framework for mapping the debug components and reconstructing their topology within a COTS platform, without running any software on the target platform and only relying on the JTAG interface. This enables precise control of debug functionalities without any reliance on the manufacturer's documentation.
3. Existing regulation algorithms, namely MemGuard, were re-engineered to operate externally via the JTAG interface. Our adaptation preserves only their core functionalities, minimizing the implementation complexity.
4. Building on the limitations of existing approaches, we introduced MemAdapt, an algorithm that combines the strengths of prior methods while addressing their shortcomings. We improved the handling of the per-period budget, taking into account the overshooting due to delays in halting and introducing the concept of budget stealing.
5. To address limitations in existing benchmarking tools, we designed LL-Mem-Stress, a highly configurable memory stressor capable of simulating diverse access patterns and able to cover a wide range of applications.

6.2. Future research

This thesis lays the groundwork for a new approach in memory bandwidth regulation, where the debug infrastructure is exploited to bring regulation to a very wide range of platforms. This allows researchers to experiment with diverse platforms, without depending on the availability of a specific development board that suits all their needs.
Newer Arm chips provide many interesting features, such as the tracing capabilities of the Embedded Trace Macrocell (ETM) or the Memory System Resource Partitioning and Monitoring (MPAM). All the popular embedded boards widely used by the research community lack these components, as they are still relatively new to the market. Our JTAG approach can help accelerate research in this regard, as it can be adapted to all platforms, as long as they allow debugging through the JTAG pins.

We have extensively used the word "JTAG" to describe our approach, even though the JTAG is only the outermost part of the debug infrastructure, only describing the access protocol through the physical interface. Future refinements aim to extend this method to cover the SWD and SWJ interfaces, to further extend the class of supported platforms.
Further investigations may focus on the electrical constraints, as well. We have provided a maximum access frequency to the JTAG debug port, but we didn't investigate the causes of the time-related constraints. Using the suitable tools (e.g., an oscilloscope) will make it possible to discover whether the restrictions come from the external physical connection, in our case, the jumper cables, or the internal JTAG circuit.

References

[1] *ARM Debug Interface v5 Architecture Specification*. ARM Limited, Cambridge, UK, Feb 2006. URL `https://developer.arm.com/documentation/ihi0031/a`. Revision A.

[2] *ARM® CoreLink™ QoS-301 Network Interconnect Advanced Quality of Service*. ARM Limited, Cambridge, UK, 2011. URL `https://developer.arm.com/documentation/ddi0451/b/`. Revision B.

[3] Alessandro Barenghi, Luca Breveglieri, Niccolò Izzo, and Gerardo Pelosi. Software-only reverse engineering of physical dram mappings for rowhammer attacks. In *2018 IEEE 3rd International Verification and Security Workshop (IVSW)*, pages 19–24, 2018. doi: 10.1109/IVSW.2018.8494868.

[4] Michael Garrett Bechtel and Heechul Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. *CoRR*, abs/1903.01314, 2019. URL `http://arxiv.org/abs/1903.01314`.

[5] *BCM2711 ARM Peripherals*. Broadcom Europe Ltd. and Raspberry Pi Ltd., Cambridge, UK, Jan 2022. URL `https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf`. Release 4.

[6] Giorgio Farina, Gautam Gala, Marcello Cinque, and Gerhard Fohler. Enabling memory access isolation in real-time cloud systems using intel's detection/regulation capabilities. *Journal of Systems Architecture*, 137:102848, 2023. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2023.102848. URL `https://www.sciencedirect.com/science/article/pii/S1383762123000279`.

[7] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-152-8. doi: 10.4230/LIPIcs.ECRTS.2020.23. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2020.23`.

[8] IEEE. Ieee standard for test access port and boundary-scan architecture. *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pages 1–444, 2013. doi: 10.1109/IEEESTD.2013.6515989.

[9] *Intel®64 and IA-32 Architectures Software Developer Manuals*. Intel Corporation, Santa Clara, CA, USA, 2020. URL `https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html`.

[10] Ivan Izhbirdeev, Denis Hoornaert, Weifan Chen, Alexander Zuepke, Youssef Hammad, Marco Caccamo, and Renato Mancuso. Coherence-aided memory bandwidth regulation. In *2024 IEEE Real-Time Systems Symposium (RTSS)*, pages 322–335, 2024. doi: 10.1109/RTSS62706.2024.00035.

[11] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real-Time Syst.*, 52(3): 356–395, May 2016. ISSN 0922-6443. doi: 10.1007/s11241-016-9248-1. URL `https://doi.org/10.1007/s11241-016-9248-1`.

[12] Tomasz Kloda, Marco Solieri, Renato Mancuso, Nicola Capodieci, Paolo Valente, and Marko Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–14, 2019. doi: 10.1109/RTAS.2019.00009.

[13] Mattia Nicolella, Shahin Roozkhosh, Denis Hoornaert, Andrea Bastoni, and Renato 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, page 184–195. ACM, June 2022. doi: 10.1145/3534879.3534888. URL `http://dx.doi.org/10.1145/3534879.3534888`.

[14] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: exploiting dram addressing for cross-cpu attacks. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, page 565–581, USA, 2016. USENIX Association. ISBN 9781931971324.

[15] Shahin Roozkhosh and Renato Mancuso. The potential of programmable logic in the middle: Cache bleaching. *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 296–309, 2020. URL `https://api.semanticscholar.org/CorpusID:214591513`.

[16] Shahin Roozkhosh, Denis Hoornaert, and Renato Mancuso. Caesar: Coherence-aided elective and seamless alternative routing via on-chip fpga. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 356–369, 2022. doi: 10.1109/RTSS55097.2022.00038.

[17] Ahsan Saeed, Dakshina Dasari, Dirk Ziegenbein, Varun Rajasekaran, Falk Rehm, Michael Pressler, Arne Hamann, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores. In *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 133–145, 2022. doi: 10.1109/RTAS54340.2022.00019.

[18] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill Companies, Incorporated, 2005. ISBN 0070570647, 9780070570641.

[19] Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. E-warp: A system-wide framework for memory bandwidth profiling and management. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 345–357, 2020. doi: 10.1109/RTSS49844.2020.00039.

[20] *UM1974 User Manual*. STMicroelectronics, Agrate Brianza (MB), Italy, Aug 2023. URL `https://www.st.com/resource/en/user_manual/um1974-stm32-nucleo144-boards-mb1137-stmicroelectronics.pdf`. Rev 10.

[21] *DS9405 STM32F427xx and STM32F429xx datasheet*. STMicroelectronics, Agrate Brianza (MB), Italy, Oct 2024. URL `https://www.st.com/resource/en/datasheet/stm32f427vg.pdf`. Rev 11.

[22] Noriaki Suzuki, Hyoseung Kim, Dionisio De Niz, Bjorn Andersson, Lutz Wrage, Mark Klein, and Ragunathan Rajkumar. Coordinated bank and cache coloring for temporal protection of memory accesses. In *2013 IEEE 16th International Conference on Computational Science and Engineering*, pages 685–692. IEEE, 2013.

[23] E. Tamura, J. Busquets-Mataix, and Antonio Martí-Campoy. Towards predictable, high-performance memory hierarchies in fixed-priority preemptive multitasking real-time systems. In *15th International Conference on Real-Time and Network Systems*, 01 2007.

[24] Prathap Kumar Valsan, Heechul Yun, and Farzad Farshchi. Taming non-blocking caches to improve isolation in multicore real-time systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12, 2016. doi: 10.1109/RTAS.2016.7461361.

[25] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64, 2009. doi: 10.1109/IISWC.2009.5306794.

[26] Andrew Wolfe. Software-based cache partitioning for real-time applications. *Journal of Computer and Software Engineering*, 2:315–327, 01 1994.

[27] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013. doi: 10.1109/RTAS.2013.6531079.

[28] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014. doi: 10.1109/RTAS.2014.6925999.

[29] Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. Parallelism-aware memory interference delay analysis for cots multicore systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 184–195, 2015. doi: 10.1109/ECRTS.2015.24.

[30] Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. Mempol: Policing core memory bandwidth from outside of the cores. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 235–248, 2023. doi: 10.1109/RTAS58335.2023.00026.

Abstract in lingua italiana

I sistemi real-time multi-core affrontano sfide significative nel garantire predicibilità degli accessi alla memoria a causa di risorse condivise come cache e DRAM. L'impredicibilità della memoria, aggravata dalle interferenze tra core, compromette l'analisi del Worst Case Execution Time (WCET), stimando limiti superiori eccessivamente pessimisti. La regolazione della banda di memoria è una soluzione promettente per mitigare l'impredicibilità della memoria nei sistemi real-time multi-core, offrendo semplicità tramite modifiche minime al software. Tuttavia, la sua adozione pratica presenta alcune criticità: l'overhead della regolazione può interferire con l'esecuzione dei task, mentre aumentare aggressivamente la frequenza per ottenere una regolazione a grana più fine può sovraccaricare le CPU. Gli approcci preesistenti si basano spesso su piattaforme eterogenee o su hardware specializzato per aggirare questi problemi, ma tali soluzioni compromettono la portabilità e scalabilità. Questa tesi introduce una soluzione innovitiva e portabile che sfrutta l'interfaccia di debug JTAG, una funzionalità hardware ampiamente disponibile, per regolare esternamente la banda di memoria, senza modificare l'hardware o il software del sistema target. Partendo da questo framework, implementiamo un algoritmo di regolazione allo stato dell'arte, MemGuard, e proponiamo successivamente MemAdapt, il nostro algoritmo di regolazione progettato specificamente per il framework. I risultati sperimentali dimostrano l'applicabilità della soluzione proposta su piattaforme ampiamente disponibili ma scarsamente documentate, come il Raspberry Pi 4 B, ottenendo una regolazione efficace con overhead minimo.

Parole chiave: sistemi real-time, multi-core, regolazione della larghezza di banda, isolamento temporale