A Centralized Performance Monitoring Architecture for Heterogeneous Multicore SoCs

Mohammed Sajjad Jafri, Abdur Rahman*, Emon Sarkar*,
Mahdi Hassen*, Gopishankar Thayyil, Ashwin Krishna Mani, Rodolfo Pellizzoni
Dept. of Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada
{msjmoham, abdur.rahman, esarkar, mhassen, gthayyil, akmani, rpellizz}@uwaterloo.ca

***Abstract*—Hardware Performance Counters (HPCs) are widely used to enable event-driven software mechanisms such as profile-guided optimization, performance analysis, and dynamic resource management in real-time systems. However, in modern embedded System-on-a-Chip (SoCs), different components - including processors, accelerators, interconnects, and memory controllers - typically implement separate and heterogeneous HPC modules. This distributed monitoring infrastructure requires multiple software interfaces and complicates the collection, synchronization, and correlation of performance data across the system.**

In this work, we propose a centralized performance monitoring architecture to efficiently collect, correlate, and process architectural events across multiple hardware components. Our design introduces Event Monitoring Units (EVUs) that capture and forward microarchitectural events to an Advanced Performance Monitoring Unit (APMU). The APMU integrates programmable counters and a specialized processing element to support flexible, event-driven software mechanisms. We implement our design for an AXI4-based system and integrate it into a RISC-V based SoC platform, which lacks advanced cross-component performance monitoring support. We demonstrate the effectiveness of our approach through case studies on real-time resource regulation, application profiling, and counter attribution.

I. Introduction

Modern embedded systems increasingly rely on heterogeneous computing architectures, combining diverse processors, multi-tiered memory hierarchies, and hardware accelerators - to meet growing performance demands. While this approach improves power and compute efficiency over homogeneous designs, it also significantly increases system complexity. In real-time domains, this complexity makes it particularly challenging to guarantee task execution times due to contention for shared resources such as caches, interconnects, I/O, and main memory. To support high-integrity systems on such platforms, sufficient run-time observability is essential. Commercial solutions [1]–[3] provide Hardware Performance Counters (HPCs) to expose low-level architectural events. These have been widely used in general-purpose and cloud systems for profiling and bottleneck analysis [4]–[6], and in embedded real-time systems to enable resource regulation mechanisms [7]–[11] that enforce timing isolation by monitoring resource usage and dynamically controlling access, e.g., by stalling cores [7], [12].

Performance counters are nowadays available not only in processors, but also in other hardware modules such as GPU [13], [14], caches [15], [16], and memory controllers [17], [18]. Each set of counters observes events local to its micro-architectural domain, e.g., the number of hits and misses to a platform-level cache made by different cores, the number of open and closed row accesses to the DRAM, the request latencies on the system interconnect, etc. However, this decentralized approach has inherent limitations. As the counters are distributed across multiple modules, the task of reconstructing system behaviour and correlating event information in different modules becomes challenging. Each set of counters often exports a different programming interface, complicating coding efforts [1]. Finally, the decentralization of HPC modules also increases the software latency for accessing the counters, which can in turn affect the performance of run-time mechanisms relying on counter values [20], [21].

In practice, run-time monitoring based on HPCs relies either on polling or on event-driven (interrupt-based) sampling. Polling is straightforward but introduces software overhead and temporal imprecision. Interrupt-based sampling can be triggered by a periodic timer, as in Linux *perf*'s `stat -I` mode [22], or by a counter overflow upon reaching a configurable threshold, the latter requiring dedicated hardware overflow support. Overflow-triggered sampling reduces temporal imprecision but suffers from interrupt skid [23], [24], i.e., a delay between event occurrence and interrupt delivery that can cause event misattribution [25]. In either case, monitoring is typically performed on the application core itself, through CPU instructions, Control and Status Registers (CSRs), or OS-level interfaces (e.g., Linux *perf* [19]), introducing overhead that may perturb the workload, a critical concern in real-time systems. Offloading monitoring to a dedicated side core [10] reduces direct interference, but still requires software-mediated reads from decentralized PMUs, incurring inter-core communication latency and synchronization overhead [20], [21].

To further reduce software involvement, modern processors provide hardware-supported tracing and sampling infrastructures, such as Arm CoreSight [26] and precise sampling extensions (e.g., Intel Processor Event Based Sampling(PEBS) [27] and Arm Statistical Profiling Extension (SPE) [28]), which capture events in hardware buffers with lower overhead and improved temporal accuracy. However, these mechanisms remain largely processor-centric: CoreSight's [26] standard

*Equal contribution; order is alphabetical.

[1]In fact, in our experience proper documentation is often missing; many hardware vendors contribute code to access HPC to the Linux *perf* tool [19], whose code often represents the only source to infer HPC behavior.

tracing components (e.g., Embedded Trace Macrocell (ETM), System Trace Macrocell (STM)) are designed to capture high-bandwidth instruction and data traces from application cores, and while the Advanced Microcontroller Bus Architecture (AMBA) Trace Bus (ATB) can in principle carry traces from other IPs, practical deployments overwhelmingly target core-level tracing; precise-event extensions such as Intel PEBS [27] and Arm SPE [28] are similarly tied to the processor pipeline. As a result, they do not natively support unified, low-latency correlation of events across heterogeneous system components such as caches, interconnects, and memory controllers, leaving system-wide observability dependent on multiple decentralized interfaces.

Event-driven run-time mechanisms in real-time systems require timely, low-overhead, and system-wide observability to enforce temporal isolation and resource regulation. Consequently, there is a need for a monitoring infrastructure that (i) minimizes software involvement, (ii) provides standardized access to events from multiple hardware components, and (iii) enables low-latency correlation and processing of performance data in a unified manner. To these ends, in this work we propose a centralized performance monitoring architecture. We provide an infrastructure where architects can instantiate custom monitoring units, called EVent Units (EVUs), in individual system components, like processors, caches, memory controllers, etc. These units are responsible for gathering architectural events and transmitting them to an Advanced Performance Monitoring Unit (APMU). The APMU collects and processes the received information: it comprises a set of configurable smart counters that can perform arithmetic and logical operations on the received event information, and a specialized instruction processor that executes software programs on the counter data. Our major contributions are:

- We propose a centralized architecture that enables unified collection of architectural events from multiple heterogeneous System-on-a-Chip (SoC) components. Our architecture is based on a standardized EVU-APMU interface to enable interoperability among event-generating components and to simplify integration of existing performance monitoring units (Section IV-A).
- We design an APMU for event aggregation, filtering, and counter updates. The APMU contains a dedicated processing element (APMU-PE) for counter processing and correlation, eliminating the software overhead, interrupt-induced perturbations, and temporal skew that affect conventional HPC-based monitoring mechanisms executing on application cores (Section IV-B). Our design provides a standardized software interface for accessing counters, ensures low-latency event collection and processing, and is both flexible and configurable.
- To prove the feasibility of our architecture, we implement in hardware description language a prototype APMU alongside multiple EVUs: (i) an AXI4 Snooping Unit (SPU), an interconnect-level EVU that monitors AXI4 transactions on the system interconnection [29], and (ii) a CVA6 Event Unit (CVA6-EVU) that extends the native performance monitoring of the RISC-V CVA6 core by exposing microarchitectural events and Program Counter (PC) markers for fine-grained event attribution and application progress tracking (Section V). We further integrate the proposed APMU and EVUs into an existing RISC-V based multicore platform lacking native advanced performance monitoring support [30], [31], and validate their functionality, resource overhead, and runtime behavior on an FPGA-based emulation platform (Section VI-A).
- Finally, we demonstrate the applicability of our architecture through two case studies: (i) a latency-based resource regulation mechanism, and (ii) fine-grained application profiling and counter attribution using PC markers (Sections VI-C and VI-D).

II. Background and Related Works

A. Hardware Performance Monitoring Counters (HPC)

Many of the popular Commercial Off-The-Shelf (COTS) and open-source computer architectures and hardware frameworks support HPCs. On the industrial front, both Intel and Arm support HPC counters in their processors [32], [33]. HPCs also form the base for many performance monitoring frameworks, such as Arm's Memory System Resource Partitioning and Monitoring (MPAM) [1] and Intel's Resource Director Technology (RDT) [2]. In the open-source domain, RISC-V [30] has adopted the "Zicntr" and "Zihpm" extensions [31], that propose a set of performance counters: CYCLE, TIME and INSTRET, which count the number of clock cycles elapsed, wall-clock real time, and the instructions retired, respectively. Apart from these, the extensions also propose 29 additional 64-bit HPCs that can be designed to count platform-specific hardware events. Unfortunately, the current list of performance monitoring events in the CVA6 family [34], [35], which is the RISC-V processor family used in our evaluation platform, is lacking. Events such as number of writebacks by the data cache and unconditional jumps, are missing. Furthermore, the RISC-V HPC specification does not mandate support for counter overflow interrupts, limiting the implementation of event-triggered sampling mechanisms on many RISC-V implementations. Another major limitation of RISC-V performance counters is that they are only accessible by the processor itself. This means that a core cannot read the counters of another core directly, which makes the execution of a multicore event-based mechanism difficult.

HPC Software Support. Core-level HPCs often vary significantly across processor generations, with different microarchitectural events supported on specific models. In some architectures, counters are not accessible at lower privilege levels, requiring traps to higher privilege modes. Combined with the limited number of available counters, this makes direct access by application programmers uncommon on general-purpose platforms. Instead, on Linux, the *perf_event* [36] kernel-level abstraction layer is used to interact with HPCs, and many libraries and tools [37]–[39] rely on it. Through *perf_event*, applications can access performance data via sampling or explicit polling mechanisms, each involving trade-offs between overhead and temporal precision. Concretely, the *perf* tool built on *perf_event* reports counters either as totals over a

workload (`perf stat`) or as periodic snapshots at a configurable interval, with a minimum interval of $1\,ms$ in current upstream kernels (`perf stat -I`). At each interval the kernel reads per-CPU counter state and user-space retrieves the deltas on cores that may also be running application threads. System-wide measurement (`-a`) is implemented as one file descriptor per monitored CPU, with software-side aggregation of the per-core results, introducing additional read overhead and temporal skew across the per-core reads [22]. PAPI [37] provides a portable cross-platform PMU API but inherits these costs, and on platforms where the underlying hardware lacks counter-overflow interrupts it falls back to a software-emulated handler driven by a periodic kernel timer. In all cases, these mechanisms execute on the cores being monitored, imposing interference that is at odds with the timing-predictability requirements of real-time workloads. Prior work [24] claims that even with overflow interrupt based specialized statistical profiling tools such as Intel PEBS, one may still suffer from extreme imprecision. While mature software stacks exist for core-level PMUs, similar abstractions are largely absent for performance counters located in other system components. We note that modern x86 server platforms do expose *uncore* PMU counters that monitor shared resources such as last-level cache slices, integrated memory controllers, and on-die interconnects via Linux `perf`, PAPI [37], or vendor performance tools such as the HPE Cray uncore subsystem [40]. However, these are organized per socket and can only be programmed from a single designated CPU per socket; event encodings differ from core HPCs and vary across socket generations, meaning cross-component correlation still requires the application to manage multiple distinct event families from specific cores. On embedded platforms, which is our target domain, no analogous uncore PMU exists, and counters in components such as platform caches, accelerators, and interconnects are typically not visible through `perf` at all. For instance, GPU performance counters are typically accessed through vendor-specific profiling frameworks (e.g., NVIDIA Nsight [41]), while counters in memory controllers, interconnects, and platform caches are often exposed through proprietary drivers, memory-mapped registers, or undocumented interfaces. Consequently, there is no standardized interface comparable to *perf_event* [36] kernel subsystem or PAPI [37] that provides a unified access model for heterogeneous counters across CPUs, caches, interconnects, and memory subsystems. Similar to *uncore* support, our architecture monitors architectural events from heterogeneous components; however, we centralized event collection on a dedicated processing element to export a unified counter access interface and improve event correlation.

HPC Usage: Off-line Approaches. HPCs provide fine-grained visibility into microarchitectural events such as instruction execution, cache behavior, branch prediction, and memory accesses [42], [43]. Prior work using HPCs can be broadly categorized into off-line and on-line techniques based on how counter data is consumed. Off-line approaches collect counter data during program execution and analyze it post-mortem to improve system understanding and performance, including performance debugging and software diagnosis [44], [45], workload profiling and program optimization [5], [6], [46], [47], power and energy analysis [48], and hardware characterization and timing analysis [49], [50].

Attributing performance-counter measurements to individual functions or code regions conventionally requires intercepting function entry and exit in software. The most direct approach is compile-time instrumentation: with GCC's `-finstrument-functions` [51], the compiler inserts calls to user-supplied hooks at function entry/exit, and the hooks then snapshot the counters of interest, e.g., through `PAPI_read` [37] or direct CSR reads. Linux *uprobes* [52], [53] provide dynamic attribution on unmodified binaries: the kernel replaces the probed instruction with a breakpoint, so that each execution of the probed address traps into the kernel and runs the attribution handler. Although the on-disk binary is untouched, the executed instruction stream is modified. In both, the cost is paid on the application core and scales with the product of invocation frequency and handler length.

HPC Usage: On-line Approaches. In contrast to off-line approaches, on-line techniques rely on real-time observation of HPC measurements to drive dynamic decisions during execution, making them inherently sensitive to counter-access latency and temporal skew. Representative examples include run-time resource management and adaptive scheduling [54]–[56] and enforcement of usage thresholds for shared hardware resources [57]. Recent works further demonstrate that HPC variations can be leveraged for real-time detection of transient execution and side-channel attacks [42], [58], and a comprehensive survey highlights the effectiveness of HPC-based monitoring combined with machine learning for resilient and real-time threat detection [59]. This trend is mirrored in industrial deployments: Intel's Threat Detection Technology (TDT) [60] drives runtime ML-based detection of cryptojacking and ransomware from CPU HPCs, and notably offloads the inference workload to the integrated GPU so that the analysis does not steal cycles from the protected application cores. This pattern, leveraging HPCs on-line while moving the processing off the application core, directly motivates the centralized, offloaded event-processing architecture we propose in this work. However, our APMU is designed as a general event processing unit that can be employed for diverse use cases, rather than targeting a specific application as TDT.

In the real-time domain, various on-line mechanisms based on HPC have been proposed to regulate accesses to shared hardware resources. MemGuard [7] implements a memory-bandwidth reservation system to prevent undue interference between cores contending for main memory access. MemGuard assigns a maximum budget in number of memory requests per regulation period to each core. If a core exceeds its budget during a period, MemGuard halts the core until the beginning of the next regulation period; this is achieved by configuring the core's HPC to raise a hardware overflow interrupt once the budget threshold is reached. Because the budget is refreshed by triggering a timer interrupt, due to overhead considerations the minimum regulation period is set to 1 ms. Among works extending the original MemGuard approach, [8] notes that contention for access to a shared LLC can also significantly affect the cores, and thus applies MemGuard to reserve LLC bandwidth rather than main memory bandwidth.

A different approach based on periodic sampling is presented in [10], [11], [61]. In particular, MemPol [10] exploits the heterogeneous architecture common in Arm SoCs to execute the regulation logic on a smaller side-core, rather than on the application cores themselves. Arm CoreSight [26], in addition to its trace components (ETM, STM), also provides a debug access interface that allows external, core-independent access to processor state and performance counters. MemPol exploits this debug access interface for cross-core memory regulation [10], using it to poll PMU counters and halt cores externally without requiring on-core software. However, this access path incurs non-trivial per-transaction latency, and being inherently polling-based, cannot react immediately to threshold crossings, leading to overshoot [10]. Our work targets this same goal of core-independent, cross-component visibility, but does so through a dedicated hardware event-packet interface rather than software polling over a debug access port, eliminating per-access latency and reducing threshold-detection delay.

B. Hardware Mechanisms for Resource Contention

Finally, several works have proposed novel hardware solutions, implemented either at the architectural simulator or RTL level, to directly track and manage resource contention. In particular, in [62]–[65], various hardware contention units are proposed, which track the amount of interference that a core suffers from every other core in the system. This includes complex resources such as coherent interconnections [65] and DRAM main memory [64]. We argue that such works are orthogonal to our approach; the discussed mechanisms could be realized as custom EVUs and the related contention information transmitted to our APMU. This could allow, for example, the implementation of smart regulation policies by combining information about contention across multiple hardware resources, similarly to how we exploit events for both LLC and main memory in Section VI-C.

Rather than relying on software-based approaches for memory regulations, some recent work have introduced custom hardware units. A Bandwidth Regulation Unit (BRU) that monitors and enforces bandwidth limits for access to LLC has been proposed in [12] and later extended in [66] to multi-bank caches. It employs a MemGuard-like replacement policy with a period of 400 cycles. The authors of [67] leverage the specific capability of AMD Ultrascale+ heterogeneous SoCs to snoop coherence traffic between LLC and main memory through FPGA logic to implement a hardware main memory bandwidth regulator. The implementation is based on a modified token bucket [68], with a lowest employed accounting period of 1 μs (1,200 cycles of the Arm application cores), and the cores are halted through CoreSight as in [10]. In Section VI-C, we show that our APMU-based regulation loop can run at comparable frequency, while retaining the flexibility of a software implementation and correlating information on access to both LLC and main memory. We further remark that the goal of our proposal is to introduce a framework for efficient event correlation and processing that can be used in diverse SoCs and across the range of HPC applications, and not just for memory regulation.

III. Challenges and Design Overview

Following the discussion in Sections I and II, we argue that existing HPC infrastructures in modern SoCs suffer from three fundamental limitations:

(C1) Monitoring overhead and interference. Conventional HPC collection relies on software mechanisms such as interrupts, syscalls, polling, or application-core reads. These introduce execution overhead, additional memory traffic, and timing perturbations to the workload under observation. Even when monitoring is offloaded to a side core, software-mediated reads of distributed counters still incur latency and synchronization overhead.

(C2) Correlation and temporal skew. Performance counters are distributed across heterogeneous hardware modules (cores, caches, interconnects, memory controllers). Reconstructing system-wide behavior requires multiple software reads at different time instants, leading to temporal skew, event misattribution, and increased latency when correlating cross-subsystem events at runtime.

(C3) Heterogeneous interfaces and fragmented tooling. As discussed in the previous section, while unified abstractions (e.g., *perf_event*) exist for core PMUs, counters in other subsystems are typically exposed via MMIO registers, proprietary drivers, or vendor-specific toolchains. This lack of a unified access interface increases integration complexity and counter-access latency for system-wide monitoring.

To address these challenges, we propose a centralized performance monitoring infrastructure composed of a set of distributed EVUs and a central APMU, as illustrated in Figure 1. Each EVU is placed close to a hardware IP block (e.g., cores, interconnects, memory controllers) to capture local architectural events and transmit structured event packets to the APMU through a standardized EVU-APMU interface. This design enables low-latency event collection while maintaining interoperability across heterogeneous IPs.

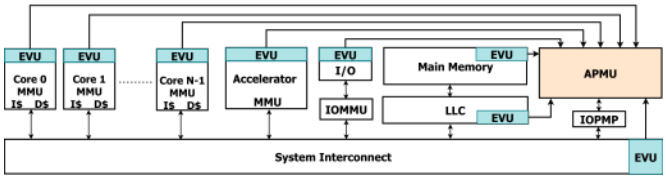

Fig. 1: A generic multicore platform with our proposed centralized performance monitoring infrastructure.

A key design principle of the proposed architecture is the decoupling of event generation from event processing. The APMU integrates programmable counters and a dedicated processing element (APMU-PE) that executes monitoring and control logic directly within the monitoring subsystem, effectively acting as a specialized side core for performance analysis. By offloading counter aggregation, filtering, and decision-making to the APMU-PE, the proposed approach avoids executing monitoring routines on application cores, thereby minimizing runtime interference and preserving timing predictability in real-time workloads.

A direct consequence of this principle, which shapes the software stack described in Section IV-C, is a *load-and-*

run programming model: monitoring policies are authored as self-contained components, compiled with the APMU-PE toolchain, and deployed onto the APMU-PE, either statically at boot or dynamically at run time rather than being expressed as application code or kernel modules. All event filtering, correlation, and control logic lives on the APMU-PE. This separation is what distinguishes our approach from conventional stacks such as Linux `perf`/*perf_event* or PAPI, in which monitoring code is either compiled into the application, patched into its binary, or executed by the OS kernel on the same cores that run the workload under observation.

This centralized and side-core-based design directly addresses the limitations of existing HPC infrastructures. First, isolating event processing within the APMU eliminates interrupt and syscall-induced overheads on application cores **(C1)**. Second, the EVU-APMU interface enables synchronized collection of events from multiple subsystems, reducing temporal skew and improving cross-component event correlation **(C2)**. Third, the unified hardware and software interface provided by the APMU abstracts the heterogeneity of component-level counters, simplifying the development of system-wide, event-based mechanisms such as resource regulation, profiling, etc. **(C3)**.

Section IV presents the architectural design of the proposed infrastructure, while Section V describes its hardware implementation and integration into a RISC-V based platform.

IV. Infrastructure Design

A. Event Unit (EVU) and EVU-APMU Interface

Each EVU encapsulates a local view of the system, abstracting component-specific signals into a unified event reporting model, and it communicates with the APMU through a standardized interface. The EVU-APMU interface is designed as a standard interface specification, analogous in spirit to Arm's AMBA AXI [29] protocol. Its purpose is to enable interoperability among event-generating IP blocks and a unified APMU framework, regardless of vendor or implementation. Each EVU is connected to a dedicated port on the APMU, enabling transmission of multiple events per clock cycle depending on the interface width and protocol. The EVU-APMU interface is divided in two layers: (a) logical layer, and (b) physical layer.

The logical layer defines the semantics of the information transmitted within an event packet. Each packet comprises three fields: (i) Event ID, (ii) Event Info, and (iii) Source ID. The Event ID uniquely identifies the type of event observed by the EVU. The Event Info field carries optional metadata associated with the event instance, such as transaction size, latency, or other event-specific attributes. The Source ID identifies the originating IP, processor execution mode, or agent responsible for triggering the event. For instance, an EVU monitoring a platform interconnect may observe the completion of a read transaction and generate an event packet containing the corresponding Event ID, along with metadata fields that specify the transaction size and completion latency. Similarly, for cache-related events, the Source ID can denote the originating master (e.g., CPU core or DMA engine) that initiated the request. In processor-based EVUs, the Source ID can also encode privilege or process context, for example, RISC-V privilege levels (e.g., user, supervisor, machine) or Arm Address Space Identifiers (ASIDs) [69].

The physical layer specifies the hardware-level signaling and connectivity between an EVU and its corresponding APMU port. Each EVU is connected to the APMU through a dedicated physical interface, which can be implemented in different forms depending on use cases and area constraints. A straightforward design employs a parallel interface, where the EVU transmits a complete event packet to the APMU every clock cycle over a set of parallel wires. While this approach allows rich event information to be transferred, including fields such as Event Info and Source ID, it may incur significant wiring overhead, especially when the packet width is large. When no event occurs, the EVU transmits a NULL event packet (or an idle code) to maintain timing alignment and ensure deterministic packet framing between EVU and APMU. Alternatively, the physical layer may employ a multi-hot encoding, where each bit in the interface corresponds to a specific Event ID. This approach allows multiple events to be signaled in the same cycle while minimizing encoding complexity. In this case, the Event Info and Source ID fields are either shared across all events or omitted when unnecessary. Designers may also define custom physical interfaces that conform to the logical-layer protocol while optimizing for area, timing, or bandwidth constraints.

Depending on the required level of flexibility, EVUs may be implemented as either fixed-function or configurable modules. In the former case, the set of architectural events to be monitored is determined at design time, which simplifies implementation and reduces area overhead. Alternatively, the EVU may support configurability at run time, allowing software to dynamically map architectural events to available Event IDs through configuration registers. This enhances observability and adaptability at the cost of potential increases in area and control complexity. In this work, we implement two representative EVU types: (i) a fixed-function AXI4-SPU that uses a parallel interface to capture transaction-level events and latency metadata (Section V-A), and (ii) a configurable CVA6-EVU that exposes a multi-hot encoded event interface for core-level micro-architectural events and PC tracking (Section V-B).

Since EVUs typically operate at the same frequency as the IPs they monitor, clock domain boundaries may exist between the EVU and the APMU. The interface supports clock-domain crossing (CDC) through asynchronous FIFOs, which buffer valid event packets and drop NULL packets. Note that event loss cannot be prevented if the long-term event production rate exceeds the APMU frequency.

B. Advanced Performance Monitoring Unit (APMU)

The APMU is the central component of the infrastructure, collecting event packets from distributed EVUs and performing on-line processing, filtering, and policy-driven actions. It comprises programmable counters that update their state based on incoming events and a lightweight processing element (APMU-PE) with local memory (Instruction and Data ScratchPad Memories (ISPM and DSPM), labeled I$ and D$

respectively, as shown in Figure 2), capable of executing control programs on the collected data, allowing both statistical monitoring and real-time policy enforcement without relying on application cores.

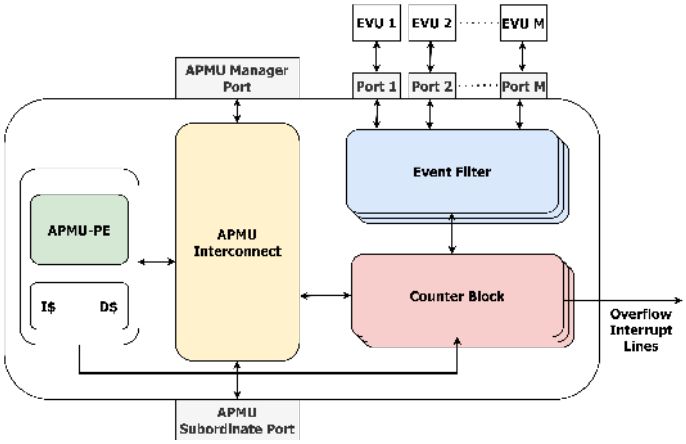

Fig. 2: APMU Block Diagram with M EVUs.

Functional decomposition. The APMU exposes two primary capabilities: (1) event aggregation & selection, (2) programmable counting & analysis. Event aggregation collects incoming packets from multiple EVU ports and applies programmable selection criteria (Event ID, Source ID, context identifiers) so that downstream logic only processes relevant data. Programmable counters support both simple counting and richer functional modes (conditional increments, adds, min/max, and range tests) on slices of the Event Info field.

Counters & Programmable Control. Each counter can (a) filter events by Event/Source/Port IDs, (b) select a bitfield within Event Info for arithmetic or logical evaluation, and (c) operate in simple-count or functional modes (accumulate, conditional increment/add, keep-min/keep-max). In simple-count mode, the counter increments each time a selected event occurs. In functional modes, the counter can manipulate the Event Info field before updating. For instance, in accumulate mode it adds the Event Info value to the counter, while keep-max/keep-min modes retain the larger or smaller value, respectively. Conditional modes allow selective updates based on comparisons (e.g., equal, less than, in-range), enabling fine-grained event evaluation. Each counter also maintains *overflow* (indicating wrap-around or saturation) and *pending* status flags; the latter is asserted whenever the counter is updated. These primitives, combined with the APMU-PE, allow software to implement flexible monitoring policies, such as per-task accumulation, latency histograms, bandwidth regulation.

Event Processing. The APMU operates on an event-driven execution model, where event packets generated by distributed EVUs are streamed into the central monitoring unit. Each incoming event is compared against the filter configuration of one or more counters, and when a match is found, the corresponding counters update their state according to the selected functional operation. This design decouples the counter functionality from the application execution on the APMU-PE, enabling event processing to occur transparently in hardware.

Wait Semantics. The APMU-PE supports synchronous event reactions through Wait-for-Pending and Wait-for-Overflow states, blocking until selected counters signal activity. When a counter status flag transitions, the APMU-PE resumes execution with a bitmap of the triggering counters, enabling immediate, context-specific handling. This event-loop model reduces software overhead and allows low-latency responses to events, such as bandwidth throttling, fine-grained statistics collection, or runtime monitoring reconfiguration.

APMU System Control Mechanism. The APMU's control capabilities stem from its system-level access and authority. It can influence the platform through two primary mechanisms. First, the APMU can raise interrupts to the application cores via the platform interrupt controller based on the status flags in each counter. Interrupts allow the APMU to influence system behavior, for example, triggering throttling actions to enforce real-time resource regulation (see Section VI-C). Second, the APMU-PE can read from and write to other components in the system through the shared interconnect via its manager port. This enables the APMU to not only report status or summary data (for example, writing counter snapshots or compact event records to a log region) but also to dynamically configure or interact with parts of the system in response to monitored conditions. Together, these capabilities make the APMU an active control agent.

Application Core-APMU Interaction. Communication between the APMU and the application cores follows a memory-mapped model over the system interconnect. Software running on the application cores can program event selectors, map Event IDs, and configure counter behaviour by writing to the APMU configuration registers, which are also accessible to the APMU-PE, allowing its control program to dynamically reconfigure counters or event mappings during runtime. Beyond configuration, the communication is bidirectional. The application cores can read and write both the APMU counters and the local memory through the APMU's subordinate port associated with the APMU-PE, enabling them to inspect monitoring data or update control parameters at runtime. The program image of the APMU-PE can be updated at runtime by writing a new binary into that memory over the system interconnect. Execution control of the APMU-PE, such as start, halt, or resume, is managed through a small set of control registers exposed in the same memory-mapped interface.

C. APMU Software Stack

Integrating the APMU into an embedded system requires a software stack that lets developers write monitoring code that runs efficiently on the resource-constrained APMU-PE, and that interoperates with the OS or hypervisor on the application cores. We therefore designed the stack around three goals: (i) make the APMU's custom instructions usable from a standard high-level toolchain, (ii) provide an event-driven runtime that lets monitoring logic react to hardware events without involving the application cores, and (iii) expose a controlled interface that lets an OS or hypervisor install, configure, and remove monitoring components at run time.

Load-and-Run Programming Model. As mentioned in Section III, monitoring code on the APMU-PE is authored as a

set of self-contained *components* rather than as application-side code. A component carries its own `init` and `exit` hooks and registers two callbacks with the runtime: an *event handler*, invoked when one of the component's monitored counters signals activity, and a *request handler*, invoked when a peer (typically an application core) sends it a service request. Components compiled into the APMU image are *static* and installed at boot; components shipped at run time (e.g., by a tenant that needs a temporary regulator or tracer) are *dynamic* and installed or torn down through the integration paths discussed below. An internal Base Component owns the request queue and routes incoming requests to the appropriate component by Request ID, isolating component logic from the queue mechanics.

Event-Triggered Runtime Model. The APMU-PE runtime is a single non-preemptive event loop built directly on the Wait-for-Pending (WFP) custom instruction. At installation, each component registers its *event bitmask*, the set of counters whose activity should invoke its event handler, and the runtime accumulates these into a global bitmask formed by the logical OR of all registered components. The loop issues WFP on this global bitmask, and the APMU-PE blocks in a low-power state until any monitored counter asserts its pending flag. WFP then returns a bitmap of the counters that triggered the wake-up; the runtime ANDs this bitmap against each component's bitmask and dispatches the event handler of every component with a nonzero match, passing the bitmap so the handler can resolve which counters fired.

Application/OS Integration. The proposed infrastructure is designed to support integration with an Operating System (OS) running in privileged mode. Address translation through the Memory Management Unit (MMU) on each application core can be used to prevent user processes from accessing the APMU and the EVU configuration registers, and from tampering with the code and data of the APMU-PE program. However, to facilitate communication between an application and the APMU, the OS can allow a specific process access to one or more APMU counters, or to a portion of the APMU data local memory, by mapping specific physical memory pages into the virtual address space of the process. For example, this might allow the application to directly notify the APMU-PE of a software event of interest by writing to a counter, to read profiling information generated by the APMU-PE, or to issue a service request to a component using that components *request handler*. Since the APMU counters are spaced out at page size boundaries in physical memory, the OS can map an individual counter to a given process without exposing the rest of the APMU. Realizing this on a general purpose OS such as Linux involves a similar approach to `perf`: a kernel module would own the APMU and enforce which counters and memory regions each process can access, while a user-space library would expose the required API.

Hypervisor Integration. Finally, the same design can also support a virtualized environment, in which case the APMU should be placed under the control of the hypervisor. The two-stage translation mechanism in the MMU can now be used to map specific APMU counters and APMU-PE local memory pages to selected Virtual Machines (VM); in this way, the guest OS for the selected VM can directly communicate with the APMU without requiring hypercalls. Similarly, the IOMMU can be used to prevent a malicious VM from using a DMA to access the APMU memory. We envision a boot process where the hypervisor loads an APMU image in the APMU local memory and then initializes the APMU-PE. For untrusted guests that should not hold direct access to the APMU, the hypervisor instead exposes a hypercall API that lets a VM install or remove dynamic components.

Security. Since the APMU can read and write to any memory location, it could tamper with the memory of the firmware/secure monitor running at a higher privilege level than the OS (e.g., machine mode in RISC-V or EL3 in Arm). For this reason, our architecture in Figure 1 includes an I/O Physical Memory Protection (IOPMP) unit, acting as a hardware checker to prevent unauthorized access to secure memory. Similarly, we can employ I/O Memory Management Units (IOMMU) to protect system memory against malicious DMA traffic; both Arm and RISC-V have corresponding specifications [70]–[72].

Comparison with Conventional Software Stacks. The above design differs from the Linux `perf_event`/`perf` and PAPI software paths. All event filtering, counter aggregation, and policy logic runs on the APMU-PE and is woken by hardware, rather than running on an application core under timer interrupts or kernel traps. Consequently the application cores see no per-sample overhead, irrespective of the effective sampling rate. The application cores only involvement in the regulation/monitoring loop is in loading the policy into the APMU-PE's instruction scratchpad.

V. Implementation

To validate the proposed architecture and demonstrate its generality, we implemented a complete RTL prototype of the centralized APMU infrastructure in SystemVerilog, including the APMU and multiple EVUs on a RISC-V SoC platform featuring the CVA6 application core [73]. The prototype is designed around standard AXI4 interconnects [74]. We implemented two types of EVUs.

The SPU is a generic EVU that monitors AXI traffic between any manager–subordinate pair without altering functionality. Leveraging the wide adoption of AXI4 across commercial and open-source systems (e.g., Arm, RISC-V, and custom SoCs), the SPU provides a natural observation point for system activity, capturing interactions among multiple IPs without modifying their logic. To demonstrate that our architecture can also leverage existing hardware monitoring support, we connect the CVA6's native performance monitoring unit to the APMU through a lightweight EVU wrapper, the CVA6-EVU, without redesigning the underlying monitoring logic.

A. AXI4 Snooping Unit

The SPU monitors AXI4 traffic by routing the original bus through its logic, as shown in Figure 3. It tracks a set of events corresponding to transaction-level activity. In particular, read transactions are tracked through the address read (AR) and read data (R) channels, while write transactions are tracked

through the address write (AW), and write response (B) channels (the write data (W) channel is not monitored in our implementation). Request events are identified from address channels (AR/AW), whereas response events are derived from the corresponding data and response channels (R/B). Each event is assigned a unique event ID and can carry additional metadata, such as transaction size, alignment, or response latency. To capture timing-related information it measures the latency of read and write responses by tracking the time between the request issuance and the corresponding response completion. Each event type in the SPU is handled by a dedicated pipeline. The resulting event packets are stored in a FIFO queue, which forms the output interface to the APMU.

A summary of the monitored events and their associated metadata is provided in Table I. Through its parallel interface, the SPU delivers events with latency-sensitive information to the APMU in real time, enabling accurate monitoring. To accurately compute response latency and maintain correct ordering for out-of-order AXI4 transactions with reused IDs, the SPU employs a modified content-addressable memory (CAM) that creates entries for each new request and removes them upon completion.

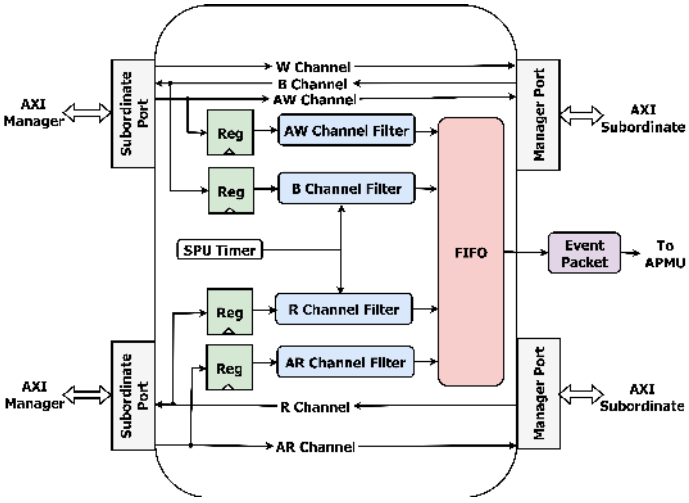

Fig. 3: SPU Block Diagram.

Event	Event Info
No event	NA
Read request Write request	Size of transaction, unaligned transfer
Read response Write response	Request latency, clock spent in contention

TABLE I: AXI4 SPU Event Table

B. CVA6 Event Unit (CVA6-EVU)

The RISC-V CVA6 core [73] is used as our application core. The CVA6 core exposes its native PMU signals as micro-architectural event outputs. Our lightweight EVU wrapper simply interfaces with such signals, selects a subset of events, and packages them into EVU-APMU event packets. We also show that similar logic can be employed to interface with existing pipeline signals to track executing instructions. The CVA6-EVU logic is runtime-configurable, allowing software to dynamically select which events and instructions to monitor.

Event ID Matching and Routing. This logic continuously monitors a set of incoming performance event signals sourced from the CVA6 hardware performance counters. Each event is uniquely identified by an Event ID. A multi-channel MUX-based selector enables the EVU to track multiple programmable events in parallel. Bitwise matching logic compares incoming signals against these Event IDs, asserting a dedicated status wire for each detected event. This design allows concurrent monitoring of a set of architectural events, including branch mispredicts, pipeline stalls, and exceptions. To extend monitoring to application-level progress, a custom Event ID is reserved for PC markers tracking.

PC Markers. This block interfaces with the commit PC and commit acknowledge signals from the CVA6 pipeline. For each committed instruction, the EVU compares the commit PC against a programmable set of reference addresses corresponding to key program locations (e.g., function entry points, basic-block boundaries, or critical code regions). When a match occurs, a dedicated progress-hit signal is asserted. The logic is designed to support multiple commit instructions. Tracking these PC markers enables fine-grained observation of application progress and phase behavior at run time; we demonstrate precise counter attribution in Section VI-C.

Detected performance events and PC marker statuses are transmitted via event packets, each consisting of three logical fields: Event ID, Event Info, and Source ID. The EVU employs a multi-hot encoding scheme at the physical interface, enabling multiple event signals and PC markers to be conveyed in a single packet per cycle. The implementation employed in our evaluation supports four event signals and four milestone addresses. The four performance events are encoded in the Event ID field, while the four PC marker bits occupy the lower portion of the Event Info field. This compact encoding ensures high-throughput, low-latency monitoring without requiring multiple packets.

C. Advanced Performance Monitoring Unit (APMU)

We employ the Ibex RISC-V core [75] as our APMU-PE, configured as a 32-bit RV32IMC processor with a two-stage pipeline (IF and ID/EX), no instruction cache or branch predictor, and a multi-cycle arithmetic unit that takes 3-4 cycles for multiplication.We employ dedicated instruction and data scratchpad memories (ISPM and DSPM) for local code execution and data storage. Instruction fetches from the ISPM complete in one cycle, while LSU accesses to the DSPM take two cycles. To support the monitoring functionality, the APMU-PE extends the base RISC-V ISA with a small set of custom control instructions. The Counter Read and Counter Write instructions provide direct access to APMU counters through a dedicated datapath, completing in two cycles without using the memory interface. The Wait-for-Pending and Wait-for-Overflow instructions monitors up to 32 counters at a time, corresponding to the processor register width. Note that this restricts the number of APMU counters to a maximum of 32. These instructions are implemented using the previously

unused opcode `7'000_0111`, with distinct function codes assigned for each operation.

Internally, the APMU uses an AXI4-Lite crossbar, adapted from the PULP AXI IP library [74], to connect the APMU-PE, local memories, and counter banks. The crossbar also provides external connectivity through both manager and subordinate AXI4-Lite interfaces. The subordinate port allows system software to configure counters and memory regions, while the manager port enables the PE to initiate data transactions.

VI. Evaluation

In this section, we first describe the integration of our developed APMU, AXI4 SPUs, and CVA6-EVUs in a RISC-V-based platform, and report implementation results for an FPGA emulator in terms of resource consumption. Since the emulator runs at low frequency due to FPGA limitations, we also provide frequency results for an ASIC implementation. We then demonstrate the applicability and benefits of our architecture using two case studies.

A. Hardware Evaluation Platform

For our evaluation, we use a modified variant of the open-source PULP (Parallel Ultra Low Power) [76]–[80] platform, emulated on the AMD Virtex UltraScale+ FPGA VCU118 board [81] as shown in Figure 4. The original PULP platform included a dual-core RISC-V CVA6 processor [73]. For our experiments, we extended it to a quad-core configuration, enabling all four cores to execute concurrently. Each core has a private 8 KiB 2-way I-Cache and a 16 KiB 8-way D-Cache (16-byte cachelines, write-back policy). The Culsans unit [82] ensures coherence across the private L1s of the four CVA6 cores. The Culsans manager port is connected to a full AXI4 crossbar, which serves as the primary interconnect fabric of the platform. The crossbar is connected to the PULP platform-level Last-Level Cache (LLC) [83], which we have extended to support a pseudo-LRU replacement policy and way partitioning, with partitioning registers exposed on the AXI4-Lite crossbar for software-controlled allocation among cores. The LLC, in turn, connects to an off-fabric DRAM. The LLC has a total capacity of 1 MiB, organized as 1024 sets with 16 ways and 64-byte cachelines. It is a write-back, write-allocate cache. The platform employs a debug module based on the SiFive debug specification 0.13 [84].

Each CVA6 core includes its own internal CVA6-EVUs, whose configuration registers are memory-mapped through the AXI4-Lite crossbar. Five external AXI4-SPUs have been added: one at the output of each core (core-to-LLC SPU), connected to a subordinate port of the full AXI4 crossbar, and one between the LLC and main memory (LLC-to-Mem SPU). These AXI4-SPUs provide full visibility into memory traffic between the cores and the LLC, and between the LLC and main memory. The APMU is configured with a 2 KiB ISPM / 16 KiB DSPM and is connected to the AXI4-Lite crossbar.

When synthesized to the targeted FPGA device, our APMU implementation can reach a maximum operational frequency of 125 Mhz. However, due to limitations in the emulation platform, the entire system, including the cores, LLC, EVUs and APMU, is clocked at 20 MHz. Note that in comparison, the off-fabric DRAM runs at 650 MHz. As a result, we acknowledge that most of the inter-core interference for access to main memory arises from how LLC misses are processed in the cache, transferred, and queued in the memory controller, rather than from contention in the DRAM memory device itself. For the same reason, we cannot meaningfully distinguish between contention for access to the same DRAM bank and to different banks [85]. FPGA resource utilization is presented in Table II. The performance counter block, which instantiates the CVA6-EVU, occupies only 0.05% of LUTs and 0.16% registers in total, of which the CVA6-EVU itself accounts for 0.02% of LUTs and 0.04% registers. The full APMU accounts for 4.49% of LUTs, of which the counter bank contributes 2.11% and the APMU-PE 1.08%.

IP Block	LUTs	Regs	BRAM	DSPs
Platform total	*773,266*	*342,276*	*6,100*	*115*
Core Cluster	75.90%	61.04%	6.31%	93.91%
One CVA6 core	7.15%	8.20%	1.49%	23.48%
Perf. Counters	0.05%	0.16%	—	—
CVA6-EVU	0.02%	0.04%	—	—
APMU	4.49%	2.66%	0.02%	0.87%
Counters & Regs	2.11%	1.50%	—	—
APMU-PE	1.08%	0.70%	0.02%	0.87%
One Core-to-LLC SPU	0.22%	0.47%	—	—
LLC-to-Mem SPU	0.24%	0.59%	—	—

TABLE II: Resource utilization of the platform and its primary IPs. Percentages are relative to the platform total (top row).

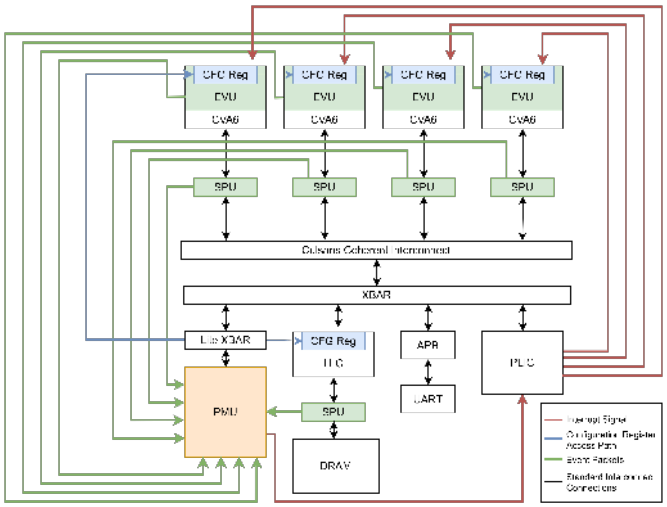

Fig. 4: Simplified view of the PULP-based platform.

ASIC Implementation Flow. Our APMU and AXI4-SPU IPs have been integrated in a similar SoC to the one employed for our FPGA emulator, which has been synthesized targeting Global Foundry 22nm node. This dual-core CVA6 implementation employs two AXI4-SPUs connected to the cores, and one to the LLC. The implementation meets a frequency of 800 Mhz for the processor block, which includes the CVA6 cores, Culsans and the two AXI4-SPUs, and 344 Mhz for the rest of the SoC. Based on post-synthesis results, neither the AXI4-

SPUs nor the APMU constraint the critical paths in the design. Due to the clock speed difference, CDC FIFOs were inserted on the interface between each AXI4-SPU and the APMU. We validated our integrated IPs based on functional simulations using memory stressor benchmarks (see next Section VI-B) and empirically found that a FIFO depth of 3 is sufficient to avoid losing events. ASIC utilization is not presented due to flattening of the hierarchy during synthesis.

B. Software/Benchmarks

We evaluate our monitoring infrastructure using both synthetic microbenchmarks and practical application benchmarks. All benchmarks are executed on bare metal without an operating system to ensure deterministic execution and eliminate OS-induced scheduling noise. We compile code for the CVA6 application cores (RV64IM) and the APMU-PE (RV32IM) using the same GCC 15.1.0 toolchain, with appropriate target configurations for each ISA. The APMU-PE custom instructions are exposed to software through inline assembly macros.

Synthetic Benchmarks. We wrote four synthetic benchmarks to stress different levels of the memory hierarchy. Each benckmark sequentially reads or writes a contiguous array with a 64-byte stride, ensuring that each access targets a unique cacheline in LLC. The LLC_r stressor performs load operations using an array size larger than the L1 D-Cache size but smaller than the LLC size; thus continuously generating misses in L1 that result in LLC read operations. LLC_{rw} uses the same array size but performs store operations. Due to the write-back L1 policy, it generates both reads and write-backs to LLC. MM_r performs load operations with an array size larger than LLC, thus causing read operations to LLC and to main memory; MM_{rw} performs store operations, generating both reads and writes to LLC and to main memory.

SDVB Benchmarks. We also evaluate a set of benchmarks from the San Diego Vision Benchmark (SDVB) suite [86], which has been used in prior work on memory regulation and real-time systems, specifically *Disparity*, *MSER*, and *Stitch* as these three workloads exhibit the highest LLC bandwidth among the SDVB suite [66]. Since we run bare metal without an operating system, benchmark binaries and input data are loaded onto a ramdisk prior to execution. Standard library functions that require system calls (e.g., `fopen`, `fread`) are replaced with custom-written wrappers that perform direct memory reads from preloaded datasets.

C. Case Study: Memory Regulation

In this case study, we evaluate the effectiveness of our monitoring framework in implementing a state-of-the-art software-based memory regulation mechanism. Our implementation uses a polling-based technique inspired by MemPol [10], where the value of relevant HPCs is read periodically every polling period P. Note that as discussed in Section II-A, our RISC-V evaluation platform's HPCs do not support overflow interrupts, making a direct comparison against MemGuard's interrupt-driven mechanism [7] infeasible. The implementation in [10] uses the L2 refill and L2 write-back counters, which provide the cumulative number of main memory reads

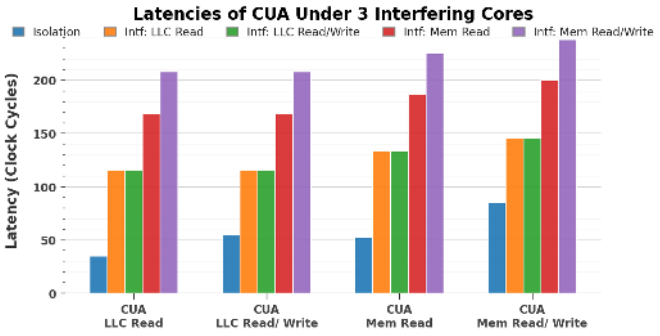

Fig. 5: Contention Results: Synthetic Benchmarks. Latency is computed as benchmark execution time divided by the number of load or store instructions.

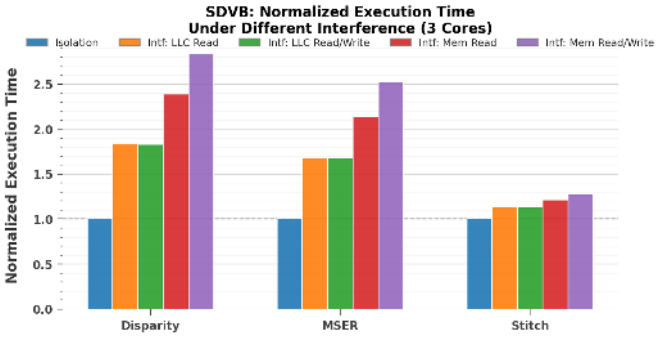

Fig. 6: Contention Results: SDVB Benchmarks (sim input). Execution time normalized by the isolation case.

$PMU_i^r(t)$ and main memory writes $PMU_i^w(t)$ performed by core c_i at time t. MemPol then computes the *cumulative memory activity* of the core as $A_i(t) = \alpha_r \cdot PMU_i^r(t) + \alpha_w \cdot PMU_i^w(t)$, where α_r and α_w are coefficients that represent the relative impacts of read and write requests on the memory resource. Each core is assigned a budget A_i^{budget}, which represents the maximum allowed memory activity per period. At run-time, the MemPol algorithm uses a sliding window approach with a configurable size w. If the cumulative memory activity of c_i increases more than $w \cdot A_i^{budget}$ in any window of w polling periods, then the core is halted. Because MemPol only polls the counters and can detect budget overruns every P time units, a core might overshoot its target budget. The second factor that can contribute to overshoot is the delay D between reading the counters and the core being stopped after a budget overrun. Smaller values of P and D decrease the maximum overshoot and thus the amount of time that a core can be halted.

Memory contention in the FPGA emulator. To determine how to best implement memory regulation, we performed a set of experiments. We run either a SDVB benchmark or one of the four synthetic stress tests on the *core under analysis* (cua). For each benchmark, we run five tests: first, we run the benchmark in isolation with no interfering cores; then, we evaluate the effects of running one of the synthetic stress tests on the other 3 cores. In all tests, we partition the LLC equally among the four cores by assigning 4 private ways (256 KiB) per core; this prevents interfering cores from evicting cache lines of the cua from LLC.

Results in Figure 5 and Figure 6 show that the cua can be significantly delayed due to LLC contention: despite LLC partitioning, requests by all cores share internal queues, read/write units, and the hit/miss unit. The LLC_r and LLC_{rw} stressors appear to be causing similar interference, likely due to the parallelism between the read and the write unit in the PULP platform-level LLC. In all cases, interfering main memory traffic causes additional interference over LLC hits, with MM_{rw} stressors causing more interference than MM_r: requests that miss in the LLC causing a dirty eviction must be processed sequentially by the LLC evict and refill units [83].

Algorithm design. Experimental results reveal that simply capturing the numbers of main memory reads and writes would not be sufficient to precisely capture the effect of interfering core. Instead, we need to also capture the number of LLC reads and LLC writes generated by each core. To this end, we employ 16 counters on the APMU, 4 for each core: 2 counters count the number of either read requests or write requests reported by the LLC-to-Mem SPU, filtering by Source ID to count the requests generated by that core only. We use $PMU_i^{MM_r}(t), PMU_i^{MM_w}(t)$ to denote the values of these counters for core c_i. The other 2 counters for c_i count the number of either read requests or write requests reported by the core-to-LLC SPU; we use $PMU_i^{LLC_r}(t), PMU_i^{LLC_w}(t)$ for their values. We then compute the memory activity as $A_i(t) = \alpha_{LLCr} \cdot PMU_i^{LLCr}(t) + \alpha_{LLCw} \cdot PMU_i^{LLCw}(t) + \alpha_{MMr} \cdot PMU_i^{MMr}(t) + \alpha_{MMw} \cdot PMU_i^{MMw}(t)$.

Similar to [67], we employ a token bucket approach: every polling period, A_i^{budget} tokens are added to the bucket, and a number of tokens equal to the memory activity in that period are removed. The maximum bucket level (number of tokens in the bucket) is $w \cdot A_i^{budget}$. If the bucket level becomes negative, then the core is halted until the level returns non-negative. Since the debug module in our platform cannot be accessed from the crossbar, the APMU regulates the cores by raising interrupts through the Platform-Level Interrupt Controller (PLIC). We use a lightweight ISR that halts the core on a wfi (wait-for-interrupt) instruction and returns to the running program after exiting the wfi.

Computing the coefficients. In [10], the coefficients α_r, α_w are computed as follows: first, the authors measured the sustainable memory bandwidth $B_{sustainable}$ of the platform, i.e., the minimum bandwidth (in number of memory requests per unit of time) that can be extracted from all cores running in parallel under worst-case conditions. Each core c_i is assigned a fraction B_i of the memory bandwidth such that $\sum B_i \leq B_{sustainable}$. The resulting budget for c_i is $A_i^{budget} = B_i \cdot P$. Since on the platform employed for their evaluation the authors measured the same sustainable bandwidth for read and write requests, they set $\alpha_r = \alpha_w = 1$.

As shown in Figures 5, 6, reads and writes to LLC and MM have widely different effects in our platform. To compute coefficients $\alpha_{LLCr}, \alpha_{LLCw}, \alpha_{MMr}, \alpha_{MMw}$, we thus use the following approach: we execute on all cores one of the four synthetic benchmarks, and measure the cumulative bandwidth of LLC reads, LLC writes, MM reads and MM writes over all cores. We then compute the coefficients by equating the derived cumulative memory activity for each of the four cases:

$$\begin{aligned}
&\alpha_{LLCr} \cdot B_{LLCr}^{LLCr} = \alpha_{LLCr} \cdot B_{LLCrw}^{LLCr} + \alpha_{LLCw} \cdot B_{LLCrw}^{LLCw} \\
&= \alpha_{LLCr} \cdot B_{MMr}^{LLCr} + \alpha_{MMr} \cdot B_{MMr}^{MMr} \\
&= \alpha_{LLCr} \cdot B_{MMrw}^{LLCr} + \alpha_{LLCw} \cdot B_{MMrw}^{LLCw} \\
&\quad + \alpha_{MMr} \cdot B_{MMrw}^{MMr} + \alpha_{MMw} \cdot B_{MMrw}^{MMw}. \qquad (1)
\end{aligned}$$

In Equation 1, the subscript in bandwidth term B represents the synthetic benchmark, while the superscript represents the considered LLC or MM operation; recalling from Section VI-B that each benchmark generates a different set of operations. For similarity with [10], we set $\alpha_{LCCr} = 1$ and then derive the other 3 parameters based on Equation 1. The resulting values of the coefficients are provided in Table III. Per-core bandwidths B_i are assigned such that $\sum B_i \leq B_{LLCr}^{LLCr}$.

α_{LLCr}	α_{LLCw}	α_{MMr}	α_{MMw}
1	0	0.612068966	0.439655172

TABLE III: Calculated regulation coefficients.

Implementation results. The worst case execution time for one iteration of our implemented APMU control program is 1,039 cycles, which are 3.02 μs at the 344 Mhz frequency of the ASIC implementation. This is significantly shorter than the 5.2 μs - 2,600 cycles at the 500 Mhz frequency of the R5 side-core - reported in [10], despite the Arm R5 core being more performant than the Ibex. While part of the difference is likely due to switching from a sliding window to token bucket regulation, a main reason is the overhead for reading the counters: in [10] the authors report a mean overhead of $274\,ns$ (137 cycles) for reading each counter, compared to just 2 cycles in our platform. Our poll-control delay is similarly small at only 648 cycles, versus 1,510 cycles (3.02 μs) in [10].

Effectiveness of regulation. Figure 7 shows a regulation trace for a synthetic benchmark with $B_{ua} = 25\%\ B_{LLCr}^{LLCr}$ (17.2 tokens refilled per iteration). The trace exhibits three phases: LLC writes, then memory writes, then LLC writes again. When memory write activity increases, the number of individual counter events decreases accordingly; yet the weighted average remains constant throughout. This demonstrates that our regulation correctly accounts for the different costs of LLC versus memory traffic. The bottom panel shows token bucket behavior where we observe the token bucket reaches a lowest value of -55 tokens during the most intensive region of the trace. This max overshoot of 55 tokens is less than the total per-period bandwidth budget across all four cores (68.8 tokens), confirming that even in the most intensive region, the overshoot remains bounded within the system's total sustainable bandwidth for a single regulation period.

Figure 8 shows the effects of regulation on SDVB benchmarks, again with 25% bandwidth allocation. We report the execution time of the benchmark in isolation, as well as when running together with 3 synthetic stressors, normalized by the execution time in isolation with no regulation. Over all benchmarks and scenarios, the maximum variation in execution time is 20.7% compared to 138.9% of the unregulated case in Figure 6. This shows that our approach effectively stabilizes

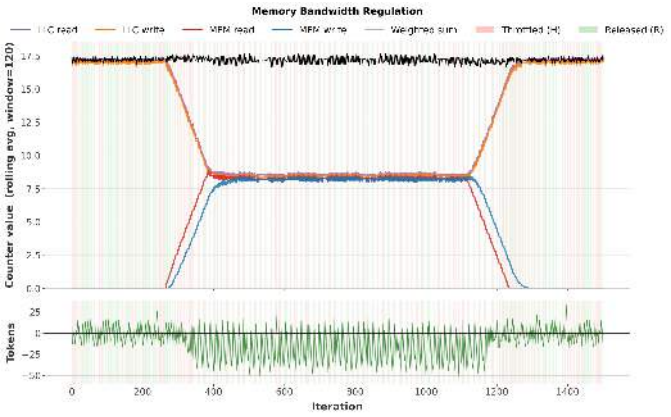

Fig. 7: Memory regulation trace under 25% bandwidth allocation (17.2 tokens/iteration). Top: counter values across three phases (LLC writes → memory writes → LLC writes). Despite shifting access patterns, the weighted average (black) remains constant. Bottom: token bucket values.

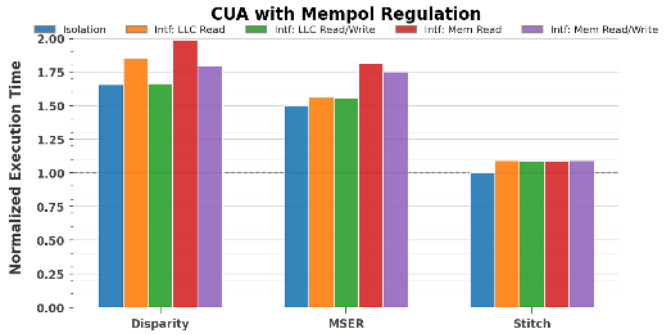

Fig. 8: Regulation results: SDVB Benchmarks (sim input). 25% Bandwidth, $w = 2, P = 2000$ cycles.

the execution time of a task under analysis independently from the activity of interfering cores.

D. Case Study: Profiling and Counter Attribution

This case study demonstrates our framework's ability to attribute hardware performance counters to specific code regions without binary modification, allowing for precise function-level profiling that isolates interference effects at fine granularity with no overhead to the application core.

We profile the SDVB disparity benchmark, which computes a stereo depth map by iterating over 8 disparity levels. Each iteration calls five functions: `padarray4()` shifts the right image, `computeSAD()` computes pixel-wise differences, `integralImage2D2D()` builds a 2D prefix sum, `finalSAD()` extracts window sums, and `findDisparity()` updates the disparity map. We target these functions to understand how microarchitectural behavior varies across them and how memory interference from co-running cores affects each. To enable function-scoped attribution, we program the CVA6-EVU with entry and exit program counters of each target function, which are easily extracted from the compiled binary's symbol table. The APMU-PE uses Wait-for-Pending to block until function entry, reads all counter values, then blocks again until function exit and reads the counters a second time; the difference yields per-function event counts. The core-to-LLC SPU concurrently tracks LLC requests using the same entry/exit deltas. The event info fields capture per-access latency; the APMU is configured to accumulate latency over each function scope. To validate this attribution mechanism, we executed synthetic tests with known expected event counts. Measured counter deltas for loads, L1D misses and LLC reads differed by at most 1 from expected values, confirming that the EVU triggers correctly scope counter increments to the intended function.

Analysis. Table IV summarizes measurements for all five functions with and without interference from three co-running cores. The key insight is a clear separation between stable and inflated metrics. Access counts (loads, stores, L1D hit ratio, and LLC reads/writes) remain virtually unchanged under interference, confirming that the workload's intrinsic memory behavior is unaffected. In contrast, timing metrics increase dramatically: cycles grow 2.7–3.8× (e.g., `padarray4()` from 0.80M to 3.02M), and LLC read latency increases 5.6–6.2× (e.g., `computeSAD()` from 0.72M to 4.34M cycles).

This separation pinpoints the interference mechanism: slowdowns stem entirely from increased memory latency due to LLC contention, not from changes in memory demand. Such attribution is possible because our hardware delimits measurements precisely to each function's execution window.

Per-Iteration Insights. Beyond aggregate interference analysis, fine-grained profiling reveals algorithmic behavior. Figure 9 shows `findDisparity()` across the 8 disparity iterations. Stores drop sharply after Cycle 1, from ∼7.5K to near zero, because the function only writes when a new minimum difference is found. This indicates most pixels converge to their best stereo match within the first two disparity levels, while later iterations perform comparisons but rarely update the output. Such insights, invisible to coarse-grained profiling, can guide optimizations like early termination.

Comparison to Software Profiling. As noted in Section II, attributing hardware counters to a specific function on a conventional software stack relies on either interrupt-based hooks, compile-time instrumentation, or periodic polling for statistical profiles, all of which interfere with the application core. As mentioned in Section II, polling-based statistical approaches are particularly imprecise: where overflow interrupts are unavailable (as with our platform), periodic polling systematically misattributes events among functions whose execution length is comparable to the polling interval.

The standard Linux approach for more precise function attribution uses *uprobes*. We run our platform on Linux kernel version 5.10.7 with uprobes backported and measure that placing uprobes at the function entry and exit adds an average latency of 34,400 clock cycles to each profiled function (excluding the actual profiling code) due to the kernel-trap overhead of processing the required system calls. Since the counters are read inside the handler the events that retire across this gap are a form of interrupt skid, charged to the target function rather than to the trap, so the captured deltas are both inflated and imprecise.

Another method is GCC's `-finstrument-functions`, which inserts entry/exit hooks into the compiled binary rather

TABLE IV: Totals of function-level hardware counters with and without memory interference (L1D hit ratio is the mean).

	No Interference							3 Cores Interference						
Function	Cycles	Loads	Stores	L1D Hit	LLC_RD	LLC_WR	RdLat	Cycles	Loads	Stores	L1D Hit	LLC_RD	LLC_WR	RdLat
Padarray4	0.80M	25.7K	25.2K	91.4%	15.3K	9.2K	0.43M	3.02M	25.7K	25.2K	91.4%	15.0K	9.1K	2.62M
ComputeSAD	1.59M	70.5K	35.3K	94.9%	25.7K	10.1K	0.72M	5.22M	70.5K	35.3K	94.9%	25.7K	10.1K	4.34M
Integral2d2d	1.79M	105K	69.8K	96.6%	23.7K	11.5K	0.66M	4.88M	105K	69.8K	96.6%	23.7K	11.6K	3.73M
finalSAD	1.23M	124K	31.1K	97.2%	15.0K	10.2K	0.43M	3.47M	124K	31.1K	97.2%	15.4K	10.3K	2.65M
FindDisp	1.17M	70.8K	17.2K	96.8%	16.5K	6.9K	0.46M	3.43M	70.8K	17.2K	96.8%	16.3K	7.1K	2.72M

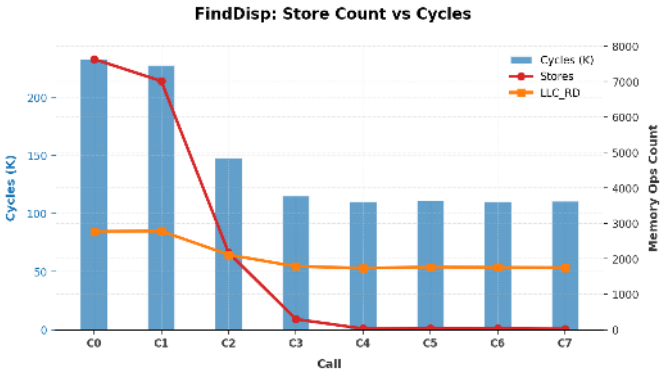

Fig. 9: Per-iteration breakdown of `findDisparity()`. Store count drops sharply after iterations C0–C1, indicating most pixels find their best stereo match early.

than trapping to the kernel. Since the binary must be modified and rebuilt based on the profiled function(s), we argue that this can be undesirable in a deployed embedded setting. Furthermore, while the method avoids the kernel-trap overhead, the inserted hooks must still read the counters inside the handler, being susceptible to skid and modifying the timing of the application based on the handler length and the function call frequency. In contrast, our mechanism described above requires no binary modification. The EVU is programmed with entry/exit PCs taken directly from the symbol table and can be retargeted to a different set of functions at runtime, with no changes to the application core and no overhead tied to handler cost or invocation frequency.

VII. Conclusion and Future Work

We presented a centralized performance monitoring architecture for multicore embedded systems, based on distributed EVUs and a central APMU that decouples event generation from processing with minimal interference on application cores. We implemented and validated the architecture on a quad-core CVA6 RISC-V platform through two case studies: a memory regulation mechanism demonstrating fast control loops with improved execution-time stability, and a fine-grained profiling mechanism enabling precise function-level counter attribution without binary instrumentation. The RTL code of all implemented hardware blocks, the assembled PULP platform and the bare-metal software stack are available open source [87], [88]. Future work includes integrating the software stack with Linux and extending it to support virtualized environments, with well-defined interfaces for OS and hypervisor interaction, as well as evaluating area and power overheads in a full ASIC implementation.

Acknowledgment

This work was supported in part by the Technology Innovation Institute, Abu Dhabi, UAE and by the NSERC. We would like to thank Davide Rossi and his group at the University of Bologna for providing support with the PULP platform.

References

[1] Memory system resource partitioning and monitoring (MPAM), for A-profile architecture. https://developer.arm.com/documentation/ddi0598/latest/.
[2] Intel Resource Director Technology (RDT) Framework. https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.
[3] AMD Xilinx. AXI performance monitor, Zynq UltraScale+ Device Technical Reference Manual (UG1085). https://www.xilinx.com/products/intellectual-property/axi_perf_mon.html.
[4] S. Kanev et al. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, 2015.
[5] A.-R. Adl-Tabatabai et al. Prefetch injection based on hardware monitoring and object metadata. *SIGPLAN Not.*, 39(6):267–276, 2004.
[6] D.-J. Oh et al. MaPHeA: A framework for lightweight memory hierarchy-aware profile-guided heap allocation. *ACM Trans. Embed. Comput. Syst.*, 22(1), 2022.
[7] H. Yun et al. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *RTAS*, pages 55–64, 2013.
[8] M. G. Bechtel and H. Yun. Denial-of-service attacks on shared cache in multicore: Analysis and prevention. *CoRR*, abs/1903.01314, 2019.
[9] N. Dagieu et al. Memguard: Memory bandwidth management in mixed criticality virtualized systems. 2016.
[10] A. Züpke et al. MemPol: Policing core memory bandwidth from outside of the cores. *Real-Time Systems*, 2024.
[11] A. Saeed et al. Memory Latency Distribution-Driven Regulation for Temporal Isolation in MPSoCs. In *ECRTS*, pages 4:1–4:23, 2023.
[12] F. Farshchi et al. BRU: Bandwidth regulation unit for real-time multicore processors. In *RTAS*, pages 364–375, 2020.
[13] S. Na et al. Common counters: Compressed encryption counters for secure GPU memory. In *HPCA*, pages 1–13, 2021.
[14] J. Barrera et al. Contention tracking in GPU last-level cache. In *ICCD*, pages 76–79, 2022.
[15] P. Roy et al. Designing secure performance metrics for last level cache. In *IPDPSW*, pages 383–392, 2023.
[16] P. P. Bhade and S. Sinha. Detection of cache side channel attacks using thread level monitoring of hardware performance counters. In *MCSoC*, pages 210–217, 2021.
[17] A. Pradhan et al. Predictable memory bandwidth regulation for DynamIQ Arm systems. In *RTCSA*, pages 126–137, 2025.
[18] J. Chen et al. CARE: Coordinated augmentation for elastic resilience on DRAM errors in data centers. In *HPCA*, pages 533–544, 2021.
[19] Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page.
[20] T. Röhl et al. Overhead analysis of performance counter measurements. In *ICPPW*, pages 176–185, 2014.
[21] D. Lo et al. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *HPCA*, pages 662–674, 2015.
[22] Linux man-pages project. `perf-stat(1)` Linux manual page. https://www.man7.org/linux/man-pages/man1/perf-stat.1.html, 2024. Accessed: 2026-06-05.
[23] B. Gottschall et al. TIP: Time-proportional instruction profiling. In *MICRO*, pages 15–27, 2021.
[24] J. Yi et al. On the precision of precise event based sampling. In *APSys*, pages 98–105, 2020.

[25] S. Das et al. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. *IEEE S&P*, pages 20–38, 2019.

[26] CoreSight architecture. https://developer.arm.com/Architectures/CoreSight%20Architecture.

[27] Intel Corporation. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C & 3D): System Programming Guide. https://cdrdv2-public.intel.com/843836/325384-sdm-vol-3abcd-dec-24.pdf, 2024.

[28] Arm Ltd. Arm Architecture Reference Manual Supplement: Statistical profiling extension, for Armv8-A. https://developer.arm.com/documentation/109429/0100/, 2017.

[29] AMBA AXI and ACE protocol specification issue f.b. https://developer.arm.com, 2017.

[30] A. Waterman et al. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. 2019.

[31] "Zicntr" and "Zihpm" counters — RISC-V extension. https://wiki.riscv.org/display/HOME/Ratified+Extensions, 2023.

[32] ARM Cortex-A72 MPCore Processor Technical Reference Manual. https://developer.arm.com/documentation/100095/0003/?lang=en.

[33] Intel 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[34] RISC-V capacity and bandwidth QoS register interface (CBQRI). https://riscv.atlassian.net/wiki/spaces/HOME/pages/16154769/RISC-V+Technical+Specifications, 2024.

[35] RISC-V quality-of-service (QoS) identifiers. https://wiki.riscv.org/display/HOME/Ratified+Extensions, 2024.

[36] L. Torvalds and Linux Kernel Contributors. Linux kernel perf events directory. https://github.com/torvalds/linux/tree/master/kernel/events, 2024.

[37] S. Browne et al. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[38] J. Reinders. *VTune Performance Analyzer Essentials*. Intel Press, 2005.

[39] Perf Wiki. Tutorial: Linux kernel profiling with perf. https://perf.wiki.kernel.org/index.php/Tutorial, 2023.

[40] Hewlett Packard Enterprise. uncore(5) — uncore performance counters. HPE Cray Programming Environment Documentation, version 24.03, 2023. https://cpe.ext.hpe.com/docs/24.03/performance-tools/man5/uncore.html, accessed 2026-06-17.

[41] NVIDIA Corporation. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems, 2024.

[42] Y. Yang et al. Exploration and exploitation of hidden PMU events. arXiv:2304.12072, 2023.

[43] A. Li et al. Tintin: a unified hardware performance profiling infrastructure to uncover and manage uncertainty. In *OSDI*, 2025.

[44] G. W. Dunlap et al. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2003.

[45] R. O'Callahan et al. Engineering record and replay for deployability. In *USENIX ATC*, pages 377–389, 2017.

[46] S. Bhatia et al. Lightweight, high-resolution monitoring for troubleshooting production systems. In *OSDI*, pages 103–116, 2008.

[47] T. A. Khan et al. DMon: Efficient detection and correction of data locality problems using selective profiling. In *OSDI*, pages 163–181, 2021.

[48] Y. Zhai et al. HaPPy: hyperthread-aware power profiling dynamically. In *USENIX ATC*, pages 211–218, 2014.

[49] D. Dasari et al. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *IEEE TrustCom*, pages 1068–1075, 2011.

[50] E. Díaz et al. Modelling multicore contention on the AURIX™ TC27x. In *DAC*, pages 1–6, 2018.

[51] GCC Team. Program instrumentation options. Using the GNU Compiler Collection (GCC), 2024. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html, accessed 2026-06-17.

[52] Srikar Dronamraju. Uprobe-tracer: Uprobe-based event tracing. The Linux Kernel documentation, 2024. https://docs.kernel.org/trace/uprobetracer.html, accessed 2026-06-17.

[53] Jonathan Corbet. Uprobes in 3.5. LWN.net, May 2012. https://lwn.net/Articles/499190/, accessed 2026-06-19.

[54] J. Fried et al. Caladan: mitigating interference at microsecond timescales. In *OSDI*, 2020.

[55] R. Gifford et al. DNA: Dynamic resource allocation for soft real-time multicore systems. *RTAS*, 2021.

[56] C. Xu et al. dCat: dynamic cache management for efficient, performance-sensitive infrastructure-as-a-service. In *EuroSys*, 2018.

[57] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, pages 109–118, 2014.

[58] J. Cho et al. Real-time detection on cache side channel attacks using performance counter monitor. In *ICTC*, pages 175–177, 2019.

[59] C. P. Chenet et al. A survey on hardware-based malware detection approaches. *IEEE Access*, 12:54115–54128, 2024.

[60] Intel Corporation. Intel Threat Detection Technology (Intel TDT). https://www.intel.com/content/www/us/en/architecture-and-technology/threat-detection-technology-brief.html, 2024. Accessed: 2026-06-05.

[61] A. Saeed et al. Memory utilization-based dynamic bandwidth regulation for temporal isolation in multi-cores. In *RTAS*, pages 133–145, 2022.

[62] J. Cardona et al. Maximum-contention control unit (MCCU): Resource access count and contention time enforcement. In *DATE*, pages 710–715, 2019.

[63] N.-J. Wessman et al. De-RISC: the first RISC-V space-grade platform for safety-critical systems. In *IEEE SCC*, pages 17–26, 2021.

[64] A. de Lecea et al. Improving timing-related guarantees for main memory in multicore critical embedded systems. In *RTSS*, pages 265–278, 2023.

[65] R. Pujol et al. Tracking coherence-related contention delays in real-time multicore systems. In *ACM/SIGAPP SAC*, pages 461–470, 2023.

[66] C. Sullivan et al. Per-bank bandwidth regulation of shared last-level cache for real-time systems. *RTSS*, pages 336–348, 2024.

[67] I. Izhbirdeev et al. Coherence-aided memory bandwidth regulation. In *RTSS*, pages 322–335, 2024.

[68] R. L. Cruz. A calculus for network delay. I. network elements in isolation. *IEEE Trans. Inf. Theory*, 37(1):114–131, 1991.

[69] Arm Ltd. Arm Architecture Reference Manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest, 2023.

[70] Arm System MMU support. https://developer.arm.com/Architectures/System%20MMU%20Support.

[71] RISC-V technical specifications. https://wiki.riscv.org/display/HOME/RISC-V+Technical+Specifications.

[72] RISC-V IOMMU architecture overview. https://open-src-soc.org/2022-05/media/slides/RISC-V-International-Day-2022-05-05-14h10-Perinne-Peresse.pdf.

[73] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, 27(11):2629–2640, 2019.

[74] A. Kurth et al. An open-source platform for high-performance non-coherent on-chip communication. *CoRR*, abs/2009.05334, 2020.

[75] Ibex: An embedded 32-bit RISC-V CPU core. https://ibex-core.readthedocs.io/en/latest/.

[76] D. Rossi et al. Energy efficient parallel computing on the PULP platform with support for OpenMP. In *IEEEI*, pages 1–5, 2014.

[77] PULP Platform. PULP platform. https://pulp-platform.org, 2014.

[78] L. Valente et al. Shaheen: An open, secure, and scalable RV64 SoC for autonomous nano-UAVs. In *IEEE Hot Chips*, pages 1–12, 2023.

[79] L. Valente et al. A heterogeneous RISC-V based SoC for secure nano-UAV navigation. *IEEE Trans. Circuits Syst. I*, pages 1–14, 2024.

[80] M. Sinigaglia et al. A heterogeneous system-on-chip with an 83 GFLOp/s, 1.2 TFLOp/s/W parallel programmable accelerator for real-time on-device learning in autonomous nano-UAVs. *IEEE Open J. Solid-State Circuits Soc.*, 2026.

[81] AMD Virtex UltraScale+ FPGA VCU118 evaluation kit. https://www.xilinx.com/products/boards-and-kits/vcu118.html.

[82] R. Tedeschi et al. Culsans: An efficient snoop-based coherency unit for the CVA6 open source RISC-V application processor. arXiv:2407.19895, 2024.

[83] PULP Platform Contributors. AXI LLC: A Parameterizable AXI4-Compliant Last-Level Cache. https://github.com/pulp-platform/axi_llc, 2025. Accessed: 2026-03-26.

[84] T. Newsome et al. *RISC-V Debug Specification, Document Version 1.0-STABLE*. 2019.

[85] H. Yun et al. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *RTAS*, pages 155–166, 2014.

[86] S. K. Venkata et al. SD-VBS: The san diego vision benchmark suite. In *IISWC*, pages 55–64, 2009.

[87] M. S. Jafri et al. he-soc: Centralized Performance Monitoring on a PULP-based SoC (EMSOFT 2026 artifact). Zenodo, https://doi.org/10.5281/zenodo.21543104, 2026.

[88] M. S. Jafri et al. APMU Software: Bare-metal case-study software (EMSOFT 2026 artifact). Zenodo, https://doi.org/10.5281/zenodo.21543106, 2026.