DAAT-MCS: A Framework to Deploy, Analyse, and Auto-Tune Interference Mitigation Techniques in Mixed-Criticality Systems

Diogo Costa
Centro ALGORITMI / LASI, Universidade do Minho, Braga, Portugal

Sandro Pinto
Centro ALGORITMI / LASI, Universidade do Minho, Braga, Portugal

Abstract

The consolidation of workloads with different criticality levels onto shared multi-core platforms is widely adopted to meet stringent SWaP-C constraints, giving rise to mixed-criticality systems. This consolidation is often enabled by static partitioning hypervisors, which provide strong spatial isolation via static assignment of resources such as cores, memory regions, and devices. However, temporal isolation remains challenging because key microarchitectural resources (e.g., the last-level cache and the memory subsystem) are still shared and can induce contention-driven slowdowns and loss of predictability. Cache coloring and related mitigation techniques can reduce this interference, but their configuration is still typically performed offline via manual profiling and trial-and-error, which is time-consuming, error-prone, and highly workload- and platform-dependent. The remaining gap is therefore not the lack of mitigation mechanisms, but the lack of a systematic way to derive a deployable partition for a given workload mix under explicit timing requirements.

This paper presents DAAT-MCS, an end-to-end framework that automates cache-color assignment for static partitioning hypervisors. It consists of two components: (i) the DAAT-MCS profiler, which systematically deploys and measures a bounded set of cache partitions on real hardware to quantify interference effects in mixed-criticality consolidations; and (ii) the DAAT-MCS tuner, which uses these measurements to automatically derive the set of system-wide cache partitions that satisfy user-defined per-core performance-degradation tolerances. Together, these components turn cache-color tuning from ad-hoc trial-and-error into a repeatable, measurement-driven integration step with explicit acceptance criteria. We evaluate DAAT-MCS on an ARMv8-A Xilinx UltraScale+ ZCU104 platform across 264 workload combinations and 7 cache-partitioning configurations (1848 setups, >60 h of automated profiling). For 28 tuner settings on a subset of 154 out of the 1848 consolidations (one memory-intensive benchmark co-scheduled with 22 co-runners across 7 cache partitions), totaling 4,312 tuner runs, DAAT-MCS finds no acceptable cache partition in 84.53% of cases and returns at least one deployable configuration in the remaining 15.47%, reducing the integration search space whenever feasibility exists and avoiding manual trial-and-error. All artifacts, including code, raw traces, and reproduction infrastructure, are publicly available.

2012 ACM Subject Classification Computer systems organization → Real-time systems

Keywords and phrases Auto-Tune, Cache Coloring, Interference, Contention, Mixed-Criticality Systems, Virtualization

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.18

Supplementary Material *Software*: `https://github.com/Diogo21Costa/DAAT-MCS` [9]

Funding *Diogo Costa*: Supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/2025 – Centro ALGORITMI (ALGORITMI/UM) `https://doi.org/10.54499/UID/00319/2025` and the Grant 2022.13378.BD.

1 Introduction

In recent decades, the trend toward digitization [8, 35] has reshaped numerous industries including automotive, robotics, and aerospace. This evolution has rapidly increased system complexity, with high-end embedded platforms evolving from simple single-core Microcontroller Units (MCUs) into complex heterogeneous architectures integrating multi-core CPUs and specialized hardware accelerators such as GPUs, TPUs, NPUs, and FPGAs [10, 17]. The demand for integration and efficiency has driven the consolidation of multiple workloads with differing criticality levels onto single hardware platforms to meet stringent size, weight, power, and cost (SWaP-C) requirements, giving rise to Mixed Criticality Systems (MCSs) [19] where safety-critical tasks coexist with mission-critical and best-effort workloads while preserving strict safety standards (e.g., ISO26262) [34].

Virtualization technology, particularly Static Partitioning Hypervisorss (SPHs) (e.g., Bao [27] and Jailhouse [39]) has emerged as the key foundational technology to consolidate MCSs, since they provide hard spatial isolation through static resource partitioning (e.g., memory regions and CPUs), eliminating runtime overheads and side-channel vulnerabilities while satisfying the guarantees required for safety certification [28]. Although SPHs provide full spatial isolation, temporal isolation remains challenging because several microarchitectural and platform-level resources are still inherently shared across Virtual Machines (VMs). These include not only the Last-Level Cache (LLC), the system interconnect/bus, and main memory, but also less obvious interference sources such as the interrupt controller and address-translation susbsystems, e.g., Input/Output Memory Management Unit (IOMMU), which can introduce cross-VM timing perturbations even under static partitioning [11, 12].

Over the past decade, extensive research has characterized the impact of contention generated by shared resources, which typically manifests as (i) increased execution time and (ii) reduced predictability [1, 7]. Together, these effects create significant challenges for system integrators aiming to certify timing behavior in safety-critical systems, as they can induce unpredictable delays and the increase of Worst-Case Execution Time (WCET) in MCSs. As a result, most interference-mitigation work has focused on two primary points: partitioning shared cache (commonly via cache coloring) to reduce LLC interference [24, 27], and regulating memory traffic (e.g., via memory-bandwidth reservation) to bound interference in the memory subsystem [49, 36]. Other complementary techniques exist (e.g., scheduling-level approaches and specialized hardware support), but cache partitioning and bandwidth regulation remain the most widely adopted software mechanisms on current COTS platforms.

While effective, these mechanisms are highly workload- and platform-dependent and expose low-level parameters that must be tuned for each consolidation scenario (e.g., the number of cache colors assigned to each VM for cache coloring [26, 24] or per-VM bandwidth budgets for memory bandwidth reservation [49, 36]). In practice, these parameters are still selected offline through manual profiling and iterative trial-and-error, which is time-consuming and error-prone [26, 49, 23, 13]. This manual configuration step remains a key limitation in the current state of the art: sub-optimal settings either over-provision isolation (wasting scarce shared resources and reducing effective capacity for co-runners) or under-provision it (incurring large and often workload-specific performance and predictability penalties), ultimately compromising both utilization and certification arguments [24, 26, 6]. More broadly, the need to retune these parameters whenever the workload mix or platform changes makes interference mitigation difficult to integrate as a repeatable step in the system-integration workflow [13, 23]. Despite extensive work on interference mitigation, the configuration of these mechanisms remains largely ad hoc: system integrators still lack a systematic and

reusable workflow that incorporates platform- and workload-specific interference evidence, expresses acceptability in terms of explicit per-workload constraints, and minimizes the number of user-defined parameters required to obtain a deployable configuration [26, 49, 23].

To address this gap, this work introduces an automated framework designed for both academia and industry. For academic use, it enables systematic, large-scale evaluation of interference scenarios and supports repeatable assessment of mitigation techniques; for industrial use, it can be directly integrated into mixed-criticality deployment workflows, reducing the cost of configuring and validating interference mitigation in practice. Our framework integrates a hypervisor-level profiling with an auto-tuning procedure to configure LLC cache coloring for predictable mixed-criticality consolidation. It takes as input per-VM degradation tolerances and derives the set of system-wide acceptable cache partitions via rule-based tolerance classification and constraint intersection across cores. By automating this platform- and workload-specific step, our framework turns cache-color selection from ad-hoc trial-and-error into a repeatable and explainable part of the integration workflow. We evaluate our framework on a Xilinx UltraScale+ ZCU104 platform by profiling 264 workload combinations across 7 cache partitioning configurations (1848 setups, >60,hours of automated runs). The evaluation shows that system-wide feasibility is strongly tolerance- and workload-dependent: under strict tolerances, many consolidations admit no acceptable cache partition in the explored configuration set, whereas relaxing the high-criticality tolerance increases the feasibility rate (up to 80.5%). In summary, this work makes the following contributions:

1. We design and implement an end-to-end profiling workflow consisting of (i) a hypervisor-integrated checkpoint profiler for synchronized, low-overhead trace collection and export, and (ii) a build-and-run automation pipeline that generates, deploys, and executes a finite set of cache-coloring configurations on real hardware (Section 3).
2. We introduce a cache-color auto-tuner that analyses profiling traces and automatically derives the set of system-wide acceptable cache partitions under user-defined constraints, eliminating ad-hoc trial-and-error cache-color selection (Section 4).
3. We conducted, to the best of our knowledge, one of the most extensive real-hardware evaluations of LLC cache coloring for MCSs on an ARMv8-A Xilinx UltraScale+ ZCU104 platform (1848 scenarios). We release all artifacts to (i) enable independent validation and follow-on research by the academic community and (ii) provide industry with a ready-to-use profiling-and-tuning workflow for MCS integration (Section 5).

2 Background and Motivation

Modern high-end embedded platforms integrate heterogeneous compute elements and expose multiple shared resources, which makes MCSs particularly vulnerable to contention. Despite substantial academic and industrial effort, achieving strong temporal isolation on Commercial Off-The-Shelf (COTS) multi-core platforms remains challenging [5, 38, 44]. In this section, we first recap the cache organization concepts required to understand cache coloring and its isolation properties, and then motivate the need for automated cache-partition tuning using empirical observations from our experimental setup, highlighting how interference and timing behavior can vary significantly across cache-color allocations and co-runner combinations.

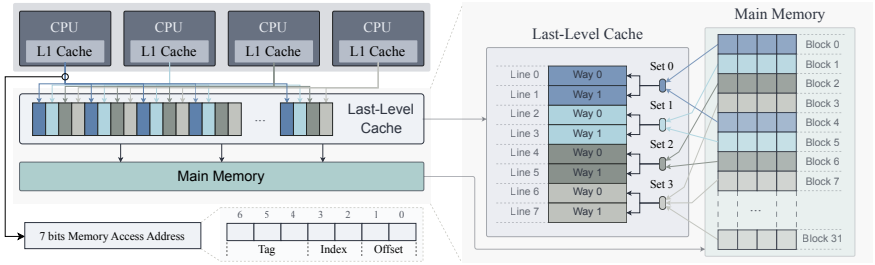

Figure 1 High-end embedded platform with shared LLC with a set-associative cache organization.

2.1 Cache Organization and Cache Coloring

Multi-level caches are used to hide memory latency and sustain throughput for increasingly complex workloads. In shared-cache multi-core systems, cache placement constraints determine where a given memory block can reside and therefore influence both performance and interference patterns. While cache organizations range from direct-mapped to fully associative, modern processors predominantly adopt set-associative designs [21].

Set-associative caches are organized as S sets, each with W ways, where each way stores one cache line of size B bytes. A cache line is the minimum transfer unit between memory and cache and contains a contiguous memory block of B bytes (typically 32 or 64 bytes). Physical addresses are commonly decomposed into: (i) an offset o, selecting a byte within a cache line, (ii) an index i, selecting the target set, and (iii) a tag t, identifying which memory block is currently stored in a given way, as depicted in Figure 1. On each memory access, the cache uses i to select the set and compares the stored tags against t; on a hit, the data is served from the matching way, while on a miss the corresponding line is fetched from main memory and placed in the selected set according to the cache's replacement policy [3, 22]. While set associativity improves average hit rates, shared LLCs introduce contention when multiple workloads/VMs compete for a limited number of ways within the same LLC indexing domain [25]. When co-runners frequently access memory regions that map to overlapping LLC indices, their combined working sets can exceed the effective cache capacity available to those indices. This leads to frequent evictions and reloads (conflict-driven thrashing), increasing access latency and undermining timing predictability. The effect is particularly problematic in mixed-criticality deployments, where high-criticality tasks can experience large timing deviations due to best-effort co-runners sharing the same LLC [26, 6, 4].

Cache Partitioning. To mitigate LLC interference, cache partitioning techniques aim to enforce spatial isolation in shared caches. Two common approaches are: (i) cache locking, which reserves specific cache lines for critical workloads [30, 37], and (ii) cache coloring (page coloring), which restricts physical page allocation to selected 'colors' so that the workload's physical addresses cover only a controlled subset of LLC sets [48, 27]. Cache locking typically requires dedicated hardware support and is not consistently available on modern high-end embedded platforms [10], making cache coloring a widely used software-only alternative. Cache coloring leverages the overlap between cache set index bits and the physical page number bits (often referred to as page color bits): by constraining page allocation to selected colors, the OS or hypervisor can control which cache sets are reachable by a workload's physical pages and thus enforce spatial isolation in the shared LLC. For simplicity, we use the toy cache organization in Figure 1 to illustrate the concept: colors correspond to groups

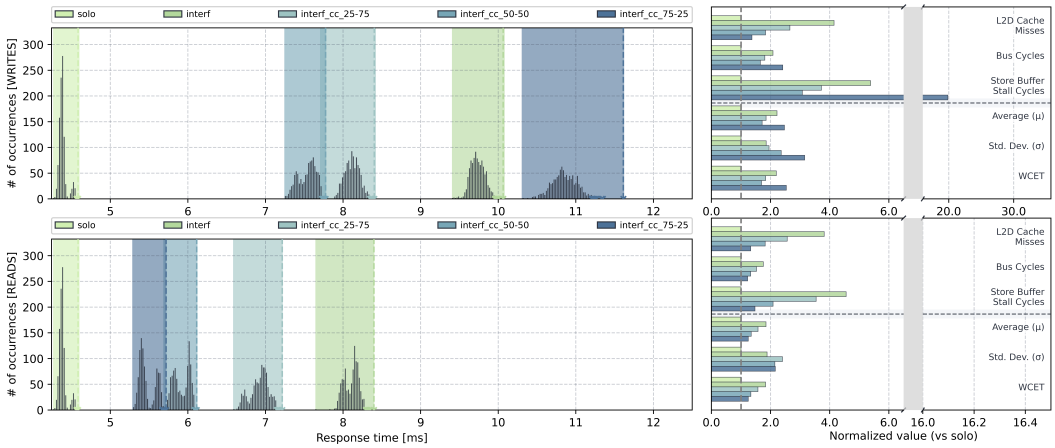

Figure 2 Distribution of response times and summary metrics for different cache partitions over 1000 executions, for LLC-sized read-intensive and write-intensive co-runners.

of sets that are indistinguishable at page-allocation granularity. In real systems, page size P is much larger than the cache line size B (e.g., 4 KiB pages and 64 B lines), so each physical page spans multiple cache lines and therefore covers multiple cache sets.

2.2 Problem and Motivation

A key challenge in integrating interference-mitigation techniques in MCSs is the configuration of these mechanisms. Cache coloring requires selecting per-VM color allocations, while memory-bandwidth reservation requires choosing enforcement periods and per-VM budgets; in both cases, the parameter choices are highly workload- and platform-dependent, and sub-optimal settings either waste shared resources or fail to bound interference. As a result, these parameters are still typically tuned offline through manual profiling and iterative trial-and-error, which is time-consuming and error-prone. Moreover, microarchitectural indicators available through performance counters (e.g., via the Performance Monitor Unit (PMU)) provide only partial visibility into end-to-end interference effects and therefore do not reliably guide configuration decisions in isolation [13].

To better understand the source of interference in our platform, we conducted a limited set of preliminary and deliberately simple experiments. Specifically, we consider a two-VM setup running on top of the Bao hypervisor [27]: one VM runs the memory-intensive *susan-c small* benchmark from the MiBench Automotive suite [18] on a dedicated CPU, while a second VM runs a custom baremetal stressor over a buffer sized to the LLC. To expose different interference regimes, we evaluate both read-intensive and write-intensive stressors and five configurations: *solo* (benchmark in isolation), *interf* (shared LLC without partitioning), and three cache-coloring partitions, *interf_cc_25-75*, *interf_cc_50-50*, and *interf_cc_75-25*, in which the benchmark receives 25%, 50%, and 75% of the LLC colors, respectively. Figure 2 summarizes the resulting response-time distributions and selected PMU-derived metrics over 1000 executions. The purpose of this experiment is not to attribute the observed slowdown to a single microarchitectural structure, but to determine whether it is explained by LLC eviction-driven contention alone or by broader memory-subsystem contention.

PMU interpretation and scope. To characterize the interference regime, we track three PMU events that expose complementary portions of the access path: (i) `l2d_cache_refill` is used as an indicator of eviction-driven reload pressure, (ii) `bus_cycles` as a coarse indicator of pressure on the shared bus and downstream memory-system activity, and (iii) `stall_sb_full` as an indicator of write-path backpressure associated with store-buffer saturation. Importantly, these events do not uniquely identify the exact contention point. In particular, the PMU events available on the target platform do not allow us to distinguish whether slowdown beyond the LLC is caused primarily by shared cache-controller structures (e.g., Miss Hold Status Registers (MSHRs) or writeback buffers), the interconnect, or the main-memory controller. Therefore, in the remainder of this section we use the term *memory subsystem* to refer collectively to these deeper shared resources beyond cache-line conflicts [13, 6].

Shared LLC without partitioning exposes eviction-driven interference. Figure 2 shows that sharing the LLC without partitioning (*interf*) compromises the temporal isolation of the benchmark in both co-runner scenarios. With the write-intensive co-runner, the mean execution time increases from 4.39 ms to 9.74 ms (2.22×), the WCET increases from 4.58 ms to 10.06 ms (2.20×), and `l2d_cache_refill` increases from 11,764 to 48,785 (4.15×). With the read-intensive co-runner, the mean execution time increases from 4.39 ms to 8.11 ms (1.85×), the WCET from 4.58 ms to 8.40 ms (1.83×), and `l2d_cache_refill` from 11,764 to 44,880 (3.82×). This joint increase in timing overhead and refill activity is consistent with conflict-driven LLC interference when the LLC is shared without partitioning.

Cache coloring initially reduces LLC eviction pressure. Figure 2 also shows that cache coloring can reduce the benchmark's execution-time overhead, which is consistent with reducing evictions of shared cache lines. For the read-intensive co-runner, increasing the benchmark's cache share consistently improves timing: relative to *interf*, mean execution time decreases by 14.65% in *interf_cc_25–75*, by 27.06% in *interf_cc_50–50*, and by 32.55% in *interf_cc_75–25*, while `l2d_cache_refill` decreases by 32.78%, 52.34%, and 65.21%, respectively. A similar trend appears initially in the write-intensive case: relative to *interf*, the mean execution time decreases by 16.64% in *interf_cc_25–75* and by 22.64% in *interf_cc_50–50*, while `l2d_cache_refill` decreases by 36.00% and 55.85%, respectively. Up to this point, the timing behavior matches what one would expect if the dominant source of interference were LLC eviction pressure. This trend breaks in the write-intensive *interf_cc_75–25* configuration. In that case, `l2d_cache_refill` continues to decrease, falling 66.94% relative to *interf* and further improving over *interf_cc_50–50*; however, the mean execution time becomes 11.36% worse than *interf* and 43.95% worse than *interf_cc_50–50*. The WCET follows the same reversal of trend, becoming 15.39% worse than *interf* and 49.36% worse than *interf_cc_50–50*. Therefore, the slowdown in *interf_cc_75–25* cannot be explained by LLC eviction-driven contention alone: the victim suffers worse timing even though refill pressure is lower.

Evidence of a shift beyond LLC eviction-driven contention. The remaining PMU events in Figure 2 suggest that, in the write-intensive *interf_cc_75–25* case, the slowdown cannot be explained by LLC-line evictions alone. When the write-intensive co-runner is assigned the smaller cache partition in *interf_cc_75–25*, its effective cache capacity is reduced, which is consistent with degraded locality and increased miss/writeback pressure. This, in turn, is consistent with higher pressure on shared resources along the memory-access path, including

the cache-controller structures involved in serving misses and writebacks, as well as the downstream interconnect and main-memory path. Consistently, in *interf_cc_75–25*, `bus_cycles` increases by 15.85% relative to *interf* and by 45.80% relative to *interf_cc_50–50*, while `stall_sb_full` increases by 3.69× and 6.45×, respectively, despite the continued reduction in `l2d_cache_refill`. Taken together, these indicators are consistent with a shift from LLC eviction-driven contention to broader *memory-subsystem* contention: the smaller partition appears to degrade the co-runner's cache locality, increase pressure on cache-controller resources, and propagate that pressure to the shared bus and main memory. Although the PMU events available on our platform do not allow us to separate the contribution of each component, they support the interpretation that the *memory subsystem*, as defined above, becomes the dominant bottleneck in this configuration. By contrast, in the read-intensive case, lower refill pressure remains associated with better timing, reinforcing that the observed reversal of trend is specific to the write-intensive configuration and is consistent with stronger write-path and store-buffer pressure.

The problem. This motivating experiment highlights two key challenges. First, giving a critical workload more LLC colors does not always improve its average-case or worst-case timing. Second, local indicators such as LLC refill counts can become misleading once the dominant source of contention moves beyond shared-line evictions. As MCSs scale to more VMs and more diverse workloads, the configuration space grows and the underlying interference mechanisms become harder to anticipate manually. This motivates systematic, workload-aware, and automated approaches for selecting cache and bandwidth configurations[1] that preserve predictable execution.

3 End-to-end Hypervisor-based Profiler

Auto-tuning cache coloring requires profiling traces $\mathcal{T}_p(c)$ for a finite set of candidate configurations $c \in \mathcal{C}$ and cores $p \in \mathcal{P}$. To collect these traces at scale, we built an end-to-end profiling framework that automates artifact generation, deployment on real hardware, and profiling-data logging. Given a scenario set $\mathcal{S}$, a candidate configuration set $\mathcal{C}$, and a repetition count R, the profiler executes a bounded campaign of size $|\mathcal{S}| \cdot |\mathcal{C}| \cdot R$. For each tuple (s, c, r), it generates the required artifacts, deploys them to the target board, collects per-core traces $\mathcal{T}_p^{(s,c,r)}$, and stores them with execution logs for offline analysis.

Compile-time and run-time stages. The compile-time stage is driven by a YAML specification describing workload setups, guest variants, and hypervisor control parameters. The framework supports multiple setup types, including standalone VMs and co-running VM scenarios, together with cache-coloring configuration and profiling utilities. When cache coloring is enabled without an explicit configuration list, the framework enumerates valid LLC partitions for the VMs in each setup under user-defined constraints. For each resulting (setup, c) pair, it generates the corresponding hypervisor configuration and builds the required artifacts, namely the guest and hypervisor images. Although our prototype targets the Bao hypervisor, this stage is designed to remain largely hypervisor-agnostic and may generalize to other SPHs.

[1] Although the framework also supports memory-bandwidth reservation, the experimental evaluation in this paper is limited to LLC cache coloring due to space constraints.

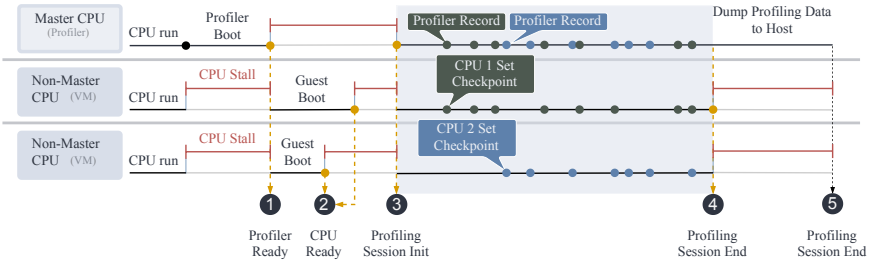

Figure 3 Run-time profiling workflow for collecting per-core checkpoint traces $\mathcal{T}_p(c)$. A profiler VM on the master core boots first and synchronizes all workload VMs; workloads then issue `PROFILE_SET_CHECKPOINT` hypercalls and the profiler records timestamps in per-core buffers. Profiling ends on `PROFILE_STOP` and traces are exported to the host.

The run-time stage iterates over the generated images, flashes the target board, boots the system, and collects the exported traces and execution logs. To support unattended campaigns over many $(\texttt{setup}, c)$ pairs, failed runs trigger a bounded reset-and-retry procedure.

Run-time profiler workflow. Figure 3 illustrates the run-time workflow. The profiler runs on a designated master core and can be enabled or disabled at boot; when profiling is disabled, this core can be assigned to a workload VM. Thus, dedicating one core to the profiler is a profiling-time design choice that simplifies orchestration and trace collection, rather than a requirement of the final deployed system. In the current implementation, the profiler runs on a lightweight Linux OS; however, the design only requires connectivity for trace export and does not depend on Linux-specific services, so it can be ported to a smaller runtime OS, e.g., Zephyr, to reduce footprint and boot time. When profiling is enabled, the hypervisor boots the profiler first and stalls the remaining cores until it reports readiness (1). Once the profiler is ready, the remaining cores boot their VMs and, after the guest OSes reach a steady state, each core issues a `CPU_READY` hypercall (2), which acts as a global synchronization barrier. The profiling session starts only after all participating VMs have reached this barrier (3), preventing boot activity from contaminating $\mathcal{T}_p(c)$. During profiling, each VM issues `PROFILE_SET_CHECKPOINT` hypercalls at developer-selected points in the workload, e.g., per iteration, at phase boundaries, or upon request completion. In Figure 3, these checkpoints appear as "Set Checkpoint" events on the non-master core timelines. Each checkpoint notifies the profiler, and the master core asynchronously records the corresponding timestamp in a per-core buffer, keeping guest-side overhead low and predictable. The session ends with a `PROFILE_STOP` hypercall (4), which finalizes the buffers and triggers export of the traces to the host framework (5). Figure 3 also shows the common case in which a designated VM, e.g., the critical one, determines when to stop profiling.

Trace semantics and scope. To ensure cross-configuration comparability of activity-based metrics, the profiling session follows a fixed termination policy. In our experiments, the designated high-criticality VM issues `PROFILE_STOP` only after completing a fixed, configuration-independent amount of work, e.g., a fixed benchmark segment or a fixed number of iterations. Because the stop condition is work-based rather than time-based, session duration may vary across configurations, e.g., due to different interference levels affecting the critical VM's completion time. Accordingly, for non-critical VMs, the checkpoint count $K_p(c)$ should be interpreted as the amount of background work completed during the execution window of the

Table 1 Notation used in Section 4.

Symbol	Meaning	Symbol	Meaning
$\mathcal{C}$	Set of cache-coloring configurations	$\mathcal{P}$	Set of cores hosting workloads
$c \in \mathcal{C}$	Single cache-coloring configuration	$p \in \mathcal{P}$	Single core
$\mathcal{T}_p(c)$	Checkpoint-duration sequence for core p under configuration c	$N_p(c)$	Number of checkpoints for core p under configuration c
$K_p(c)$	Activity metric: $K_p(c) = N_p(c)$	$T_p(c)$	Total profiled time (kcycles)
$Q_p(c)$	Tail metric: $Q_p(c) = \text{p99}(\mathcal{T}_p(c))$	$W_p(c)$	Worst observed interval time
$S_p(c)$	Per-core score used for tuning	$c_p^\star$	Baseline configuration for core p
B_p^{rel}	Relative baseline value: $B_p^{\text{rel}} = S_p(c_p^\star)$	B_p^{iso}	Isolation baseline value for core p
$rel_p(c)$	Deviation of c from B_p^{rel}	$iso_p(c)$	Deviation of c from B_p^{iso} (optional)
τ_p	Tolerance threshold for core p	$class_p(c)$	Class of configuration c for core p
α	Near-tolerance slack factor	$\mathcal{C}_{\text{acc}}$	Final set of acceptable configurations

designated high-criticality VM, rather than as absolute throughput over a fixed wall-clock interval; equivalently, it captures best-effort progress during that fixed-work execution window. Alternatively, the same mechanism can require all participating VMs to issue `PROFILE_STOP` before ending the session. In both cases, the framework produces a consistent set of per-core checkpoint traces $\mathcal{T}_p(c)$ for each evaluated configuration $c \in \mathcal{C}$. The current workflow characterizes the finite scenario set explicitly described in the campaign specification; it does not exhaustively enumerate all asynchronous phasings, event arrivals, or input-dependent execution paths. Therefore, the resulting traces should be interpreted as empirical evidence for the profiled scenarios, not as an exhaustive characterization of all possible executions.

4 Auto-Assignment of LLC Colors

The framework's tuner uses measurements to choose cache-color assignments for mixed-criticality workloads co-scheduled on a multi-core platform with a shared LLC. Rather than relying on analytical models of cache and memory behavior, the tuner operates directly on empirical profiling traces gathered for a finite set of candidate color configurations. DAAT-MCS performs (i) rule-based classification of configurations into tolerance regions (within tolerance, near tolerance, and outside tolerance), and (ii) constraint-based reasoning to derive the set of cache partitions that are suitable system-wide across all participating cores. Table 1 summarizes the notation used throughout this section.

Exploration model. In the current implementation, DAAT-MCS does not perform online search over an implicit combinatorial space. Instead, it operates on a finite candidate set $\mathcal{C}$ generated offline by the profiling workflow described in Section 3, where each $c \in \mathcal{C}$ denotes one complete system-wide cache-color assignment across the participating workloads/cores and has already been deployed and measured on real hardware. Therefore, the contribution of the tuner in this paper is not a heuristic search strategy, but a deterministic measurement-driven selection step that maps the enumerated set $\mathcal{C}$ into the subset $\mathcal{C}_{\text{acc}}$ of system-wide acceptable cache partitions. This distinction is important because the tuner itself performs filtering, classification, and ranking over measured candidates, while scalability at the full-system level is primarily determined by the offline profiling campaign required to populate $\mathcal{C}$.

Per-core trace metrics and scores. We denote by $\mathcal{C}$ the set of cache-coloring configurations under evaluation. For each configuration $c \in \mathcal{C}$ and each CPU $p \in \mathcal{P}$, we profile execution by tracing workload checkpoints to produce a duration sequence $\mathcal{T}_p(c) =$

$\{t_{p,1}(c), t_{p,2}(c), \ldots, t_{p,N_p(c)}(c)\}$, where each $t_{p,i}(c)$ is the measured duration of the i-th profiling interval on core p under configuration c, and $N_p(c)$ is the number of checkpoints observed. From this sequence, the tuner derives scalar metrics such as

$$K_p(c) = N_p(c), \qquad T_p(c) = \frac{1}{10^3} \sum_{i=1}^{N_p(c)} t_{p,i}(c),$$

where $K_p(c)$ captures activity (checkpoint count) while $T_p(c)$ captures the total profiled time in kcycles. The factor 10^3 is a unit-conversion constant used only to express the accumulated duration in kcycles rather than cycles; it keeps the reported values compact in tables and figures, but does not affect ranking, degradation, or acceptability because all comparisons are performed after consistent normalization.

As described in Section 3, when the profiling session is terminated by a designated high-criticality VM after completing a fixed, configuration-independent amount of work, $K_p(c)$ for non-critical workloads should be interpreted as progress achieved during that execution window, rather than as absolute throughput over a fixed wall-clock interval. The tuner then selects a single score metric $S_p(c)$ per core and configuration based on workload criticality. Lower $S_p(c)$ is better for higher-criticality workloads, whereas higher $S_p(c)$ is better for lower-criticality workloads.

Score instantiation. DAAT-MCS is agnostic to the specific scalar score $S_p(c)$ as long as it preserves a total order of configurations for each core p. While this work instantiates $S_p(c)$ using total profiled time $T_p(c)$ for higher-criticality cores and activity $K_p(c)$ for lower-criticality cores, the same tuning logic also supports real-time-oriented objectives derived from $\mathcal{T}_p(c)$, such as the worst-case observed interval $W_p(c) = \max_i t_{p,i}(c)$, a tail-oriented metric such as $Q_p(c) = \mathrm{p99}(\mathcal{T}_p(c))$, or a deadline-miss ratio when checkpoints correspond to jobs with deadlines. In this sense, the tuner separates (i) the collection of empirical traces from (ii) the symbolic selection policy, enabling alternative timing objectives without changing the underlying feasibility reasoning.

Per-core baselines. Given a chosen score $S_p(c)$, the tuner establishes a per-core relative baseline by selecting a baseline configuration $c_p^\star$ and value B_p^{rel}: For higher-criticality cores, $c_p^\star$ yields the best observed timing under S_p; for lower-criticality cores, it yields the highest observed activity. By default, DAAT-MCS normalizes degradation to B_p^{rel}, i.e., the best configuration within the evaluated candidate set $\mathcal{C}$. This makes the tuner rank and filter feasible partitions relative to the best observed profiled candidate, rather than relative to solo execution. When an absolute reference is required, the same formulation can be instantiated with an isolation baseline B_p^{iso} obtained from standalone execution of workload p without co-runners. The tuning logic itself remains unchanged; only the reference used to compute degradation differs. In the rest of this section, we use the relative baseline unless otherwise stated.

Quantifying degradation. Using the relative baseline, the tuner computes a degradation factor for each configuration c and core p:

$$rel_p(c) = \begin{cases} 100 \cdot \dfrac{S_p(c) - B_p^{\mathrm{rel}}}{B_p^{\mathrm{rel}}}, & \text{higher-criticality core,} \\ 100 \cdot \dfrac{B_p^{\mathrm{rel}} - S_p(c)}{B_p^{\mathrm{rel}}}, & \text{lower-criticality core.} \end{cases}$$

When an isolation baseline is available, the same form yields $iso_p(c)$ by replacing B_p^{rel} with B_p^{iso}. For higher-criticality workloads, $rel_p(c)$ is the percentage slowdown from the baseline; for lower-criticality workloads, it is the percentage activity reduction from the baseline.

Selection steps. Given the measured candidate set $\mathcal{C}$, the tuner applies the following deterministic steps: (1) select the score $S_p(c)$ for each core p; (2) compute the per-core baseline B_p^{rel} and baseline configuration $c_p^\star$; (3) evaluate $rel_p(c)$ for every $p \in \mathcal{P}$ and $c \in \mathcal{C}$; (4) classify each pair (p, c) into tolerance regions; (5) intersect the acceptable per-core candidate sets to obtain $\mathcal{C}_{\text{acc}}$; and (6) if multiple feasible configurations remain, rank them according to a deterministic secondary criterion. This makes explicit that DAAT-MCS is a measurement-driven filtering and ranking stage over an already enumerated set of candidates, rather than a search heuristic over unmeasured configurations.

Tolerance classification and constraint reasoning. DAAT-MCS treats each configuration c as a point in a per-core degradation space, with one degradation value for each core p. Tolerances are needed because, in a shared-resource setting, insisting that every workload match its per-core best observed baseline exactly would often eliminate otherwise deployable system-wide configurations; instead, the user specifies how much degradation each workload may tolerate according to its criticality. This also lets the framework represent asymmetric requirements, e.g., tight slowdown bounds for higher-criticality workloads and looser progress-loss bounds for lower-criticality ones. We additionally define a near-tolerance slack factor $\alpha > 1$ that controls the width of the near-tolerance region. In this work, and in the toy example below, we use $\alpha = 1.2$, such that degradations up to 20% above the baseline tolerance are labeled as near tolerance. Formally,

$$class_p(c) = \begin{cases} 0, & rel_p(c) \le \tau_p \qquad \text{(within tolerance)}, \\ 1, & \tau_p < rel_p(c) \le \alpha\tau_p \qquad \text{(near tolerance)}, \\ 2, & \text{otherwise} \qquad \text{(outside tolerance)}. \end{cases}$$

System-wide acceptable configurations are then obtained by intersecting the acceptable per-core sets and discarding any configuration that violates the policy on at least one core:

$$\mathcal{C}_{\text{acc}} = \bigcap_{p \in \mathcal{P}} \{\, c \in \mathcal{C} \mid class_p(c) \in \{0, 1\} \,\}.$$

This produces a set of configurations whose performance remains sufficiently close to each workload's baseline behavior, while still allowing a controlled relaxation through class 1. The same decision logic can be interpreted as a deterministic pruning tree, in which per-core constraints are applied sequentially and any configuration classified as class 2 for one core is removed from further consideration. If multiple feasible configurations remain after constraint intersection, DAAT-MCS ranks them using two ordered criteria. First, it prefers configurations with fewer cores in the near-tolerance region, i.e., fewer cores for which $class_p(c) = 1$. Second, among configurations with the same number of near-tolerance cores, it prefers the one with the lowest aggregate degradation, $\sum_{p \in \mathcal{P}} rel_p(c)$. where $n_1(c)$ denotes the number of cores classified as near tolerance. Thus, the first term is the primary ranking criterion and the second term is used only to break ties. If a higher-criticality workload already meets its deadlines for all candidates, the tuner does not attempt to "optimize past" that external requirement; rather, it still filters and ranks configurations according to the user-specified tolerance policy, or, when desired, the score $S_p(c)$ can be instantiated directly with a deadline-oriented metric.

Table 2 Toy example: configurations, degradations, and acceptability.

		c_1	c_2	c_3	c_4
p_0 (critical)	LLC allocation (p_0 share)	70%	60%	50%	40%
$\tau_0 = 5\%$	$S_0(c) = T_0(c)$ (kcycles)	120	**100**	106	130
	$rel_0(c)$ (%)	20	**0**	6	30
	$class_0(c)$	2	0	1	2
p_1 (non-critical)	LLC allocation (p_1 share)	30%	40%	50%	60%
$\tau_1 = 30\%$	$S_1(c) = K_1(c)$ (checkpoints)	50	80	60	**90**
	$rel_1(c)$ (%)	44.4	11.1	33.3	**0**
	$class_1(c)$	2	0	1	0
Acceptable on both VMs?		✗	✓	✓	✗

Toy example. To make this concrete, consider a two-core system $\mathcal{P} = \{p_0, p_1\}$ with one higher-criticality core p_0 and one lower-criticality core p_1. We evaluate four cache-color configurations $\mathcal{C} = \{c_1, c_2, c_3, c_4\}$, each assigning a different fraction of LLC colors to each core. We set tolerances to $\tau_0 = 5\%$ for the critical workload and $\tau_1 = 30\%$ for the non-critical workload, and use $\alpha = 1.2$. Table 2 reports the raw per-core scores $S_p(c)$, the derived degradations $rel_p(c)$, the corresponding classes $class_p(c)$, and the final acceptability outcome.

Step 1: Per-core performance scores and baselines. For each configuration c, the tuner computes a per-core score $S_p(c)$ based on workload criticality: the critical core uses $S_0(c) = T_0(c)$ (execution time in kcycles, lower is better), while the non-critical core uses $S_1(c) = K_1(c)$ (checkpoint count, higher is better). The tuner then establishes per-core baselines over $\mathcal{C}$ only: $c_0^\star = \arg\min_{c\in\mathcal{C}} T_0(c)$ with $B_0^{\text{rel}} = T_0(c_0^\star) = 100$ (here, c_2), and $c_1^\star = \arg\max_{c\in\mathcal{C}} K_1(c)$ with $B_1^{\text{rel}} = K_1(c_1^\star) = 90$ (here, c_4).

Step 2: Quantifying performance degradation. Given the baselines B_p^{rel}, the tuner converts raw scores into relative degradations $rel_p(c)$. For the critical core, $rel_0(c) = 100 \cdot \frac{T_0(c) - B_0^{\text{rel}}}{B_0^{\text{rel}}}$ (slowdown from the best observed time); for the non-critical core, $rel_1(c) = 100 \cdot \frac{B_1^{\text{rel}} - K_1(c)}{B_1^{\text{rel}}}$ (activity loss from the best observed checkpoint count). In this example, p_0 observes $rel_0(c_1) = 20\%$, $rel_0(c_3) = 6\%$, and $rel_0(c_4) = 30\%$, while p_1 observes $rel_1(c_1) = 44.4\%$, $rel_1(c_2) = 11.1\%$, and $rel_1(c_3) = 33.3\%$.

Step 3: Rule-based classification and constraint intersection. Finally, the tuner applies the tolerance policy to label each configuration: class 0 if $rel_p(c) \le \tau_p$, class 1 if $\tau_p < rel_p(c) \le \alpha\tau_p$, and class 2 otherwise. A configuration is acceptable only if it is not an outlier for any core, i.e., $class_p(c) \in \{0, 1\}$ for all p. For p_0, with $\tau_0 = 5\%$ and $\alpha = 1.2$, the near-tolerance threshold is 6%, so only c_2 and c_3 remain feasible, while c_1 and c_4 are rejected (class 2). For p_1 ($\alpha\tau_1 = 36\%$), c_2 and c_4 fall into class 0, c_3 falls into class 1, and only c_1 is rejected (class 2). Intersecting both per-core feasible sets therefore yields $\mathcal{C}_{\text{acc}} = \{c_2, c_3\}$.

5 Evaluation

To assess the proposed solution, the evaluation covers two complementary aspects: (i) the performance overhead of the hypervisor-based profiling framework, and (ii) the effectiveness of the cache-coloring auto-tuner in a realistic consolidation scenario. We first describe the experimental setup, workloads, cache-coloring configurations, and the metrics used throughout the section. We then quantify the cost of the profiling instrumentation via a

Table 3 Configurations and evaluation parameters with 3 VMs: one high-criticality ($HcVM$), one medium/low-criticality ($MLcVM$), and one best-effort ($BEcVM$), considering six cache-coloring partitions (`c1`–`c6`, detailed in Table 4) and one `interf` setup (no cache partitioning).

Criticality	VM	vCPUs	Workload suites	Colors/VM	Tolerance τ_p
High	HcVM	1	MiBench Automotive (12)	≥ 2	7%
Medium/Low	MLcVM	1	Embench (22)	≥ 2	12%
Best-effort	BEcVM	1	Mem.-intensive stressor (1)	≥ 2	50%

dedicated microbenchmark and report its impact for different checkpoint rates. Finally, we evaluate the auto-tuner, showing how acceptable cache partitions vary across consolidations and how those selections translate into interference reduction and improved timing behavior.

5.1 Methodology

This subsection describes the experimental context used throughout the evaluation. We first present the target platform and cache-coloring configuration space, then introduce the consolidation use case used to mimic a MCS. Finally, we detail the data-collection workflow (profiling and trace extraction) and the offline processing steps used by the tuner to compute per-VM degradation and derive acceptable cache partitions.

Evaluation setup. For simplicity and reproducibility, all experiments target a single platform/hypervisor pair: the Bao hypervisor running on an ARMv8-A Xilinx UltraScale+ ZCU104-class platform. The target SoC integrates a quad-core Arm Cortex-A53 cluster (1.2 GHz) with private L1 caches and a shared 1 MiB L2 cache, which acts as the LLC for the cluster. Although the platform supports up to 16 colors at the LLC level, our configuration uses $N_{\text{LLC}} = 8$ colors to avoid partitioning the private L1 caches.

Evaluation use case. We consider three VMs with one vCPU each (one per core): a high-criticality VM ($HcVM$), a medium/low-criticality VM ($MLcVM$), and a best-effort VM ($BEcVM$), as summarized in Table 3. These VMs represent three distinct workload roles in the consolidation: a timing-critical workload, a progress-oriented non-critical workload, and a background interfering workload.

- **MiBench VM (high criticality, $HcVM$).** Runs the MiBench Automotive suite (12 benchmarks) [18] in a Linux environment. The three most memory-intensive variants (*qsort*, *susan-corners*, *susan-edges*) are included alongside nine others. Checkpoints mark well-defined application phases; the tuner uses execution time $T_p(c)$ (kcycles) as the score.
- **Embench VM (medium/low criticality, $MLcVM$).** Runs 22 Embench benchmarks representing diverse embedded workloads. The tuner uses checkpoint count $K_p(c)$ as an activity metric, interpreted as progress achieved during the execution window of $HcVM$.
- **Memory-intensive stressor VM (best effort, $BEcVM$).** A baremetal application that continuously writes to a 1 MiB buffer (i.e., the LLC size) with a 64-byte cache-line stride. We use only the write-intensive variant, since Section 2 showed it induces a more severe interference regime than the read-intensive one. Because its working set may exceed the cache space effectively available under some partitions, it can stress not only the LLC but also downstream memory-system resources; accordingly, we refer to it as a *memory-intensive stressor*. As for $MLcVM$, the tuner uses checkpoint count $K_p(c)$, interpreted as progress achieved during the fixed-work execution window of $HcVM$.

Table 4 Mapping between cache-coloring configuration labels and per-VM LLC color allocations. Percentages are computed as $\frac{\#\text{colors}}{N_{\text{LLC}}} \times 100$, with $N_{\text{LLC}} = 8$.

VM	**interf**	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**
HcVM	–	2 (25.0%)	2 (25.0%)	2 (25.0%)	3 (37.5%)	3 (37.5%)	4 (50.0%)
MLcVM	–	2 (25.0%)	4 (50.0%)	3 (37.5%)	2 (25.0%)	3 (37.5%)	2 (25.0%)
BEcVM	–	4 (50.0%)	2 (25.0%)	3 (37.5%)	3 (37.5%)	2 (25.0%)	2 (25.0%)

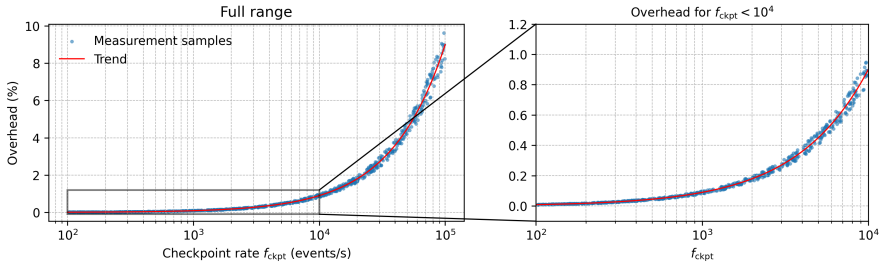

Figure 4 Checkpoint overhead as a function of checkpoint rate (from 100 to 100k checkpoints/s).

Configuration space and experiment workflow. The evaluation proceeds in two stages. First, the profiling framework enumerates all workload combinations and deploys each setup to the target board, collecting checkpoint traces for every cache-color configuration. Second, the DAAT-MCS tuner processes the dataset offline to compute, for each CPU p, a baseline score and the relative degradation $rel_p(c)$ induced by configuration c; acceptable configurations are then derived under a tolerance vector τ. We only consider cache-color partitions where each VM receives at least two LLC colors, yielding six feasible colorings for the 3-VM setup; these are reported as `c1`–`c6`, while `interf` denotes the no-coloring baseline. The lower bound of two colors per VM is imposed to avoid degenerate partitions with extremely limited effective cache reach and to keep the explored candidate set aligned with practically deployable configurations on this platform. Across the full campaign, the framework evaluates $12 \times 22 \times 1 = 264$ workload combinations. For each (workload combination, c), measurements are obtained from 1000 repeated executions; unless stated otherwise, the reported scalar score for that point is derived from the corresponding profiling trace collected in that configuration.

Metrics and baselines. For each CPU p and cache configuration c, we compute a per-workload score $S_p(c)$ as defined in Section 4: $T_p(c)$ for the high-criticality VM on $HcVM$, and $K_p(c)$ for the remaining VMs on $MLcVM$ and $BEcVM$. For non-critical VMs, $K_p(c)$ is an activity/progress metric collected over the critical VM's fixed-work execution window. DAAT-MCS's primary objective is feasibility: to derive the set $\mathcal{C}_{\text{acc}}$ of system-wide acceptable configurations under τ via intersection of per-core acceptability constraints (Section 4). In this evaluation, $\mathcal{C}$ is a finite, enumerated candidate set (here, $\{\texttt{interf}, \texttt{c1}, \ldots, \texttt{c6}\}$), and the tuner is assessed on its ability to filter and explain feasible choices within $\mathcal{C}$, rather than on search efficiency over unmeasured configurations. Unless stated otherwise, degradations $rel_p(c)$ are computed relative to the best configuration within the evaluated candidate set $\mathcal{C}$ for each consolidation instance (selection-centric baseline), and are used to derive acceptability classes; this relative baseline is used only for normalization within $\mathcal{C}$ and does not constitute an isolation reference point. To capture interference bursts that may not be visible in aggregate time, we additionally report tail checkpoint-interval metrics for the critical VM on $HcVM$: the p99 and the maximum observed interval duration derived from $\mathcal{T}_0(c)$. For a

fixed MiBench workload and configuration, tail metrics are computed by pooling checkpoint intervals across the 22 co-runner setups (i.e., concatenating $\mathcal{T}_0(c)$ across co-runners before computing p99/max).

Scope of the profiling campaign. The evaluation targets fixed workload instances and controlled co-runner combinations generated by the framework, and therefore provides a systematic characterization of the explored candidate set rather than exhaustive coverage of all asynchronous arrivals, event-triggered activations, or dynamic execution paths. This design matches the intended use of DAAT-MCS as an offline integration-time tool for deriving deployable cache partitions from measured workload mixes.

5.2 Instrumentation Overhead

We quantify the overhead of checkpoint-based profiling by directly measuring the runtime cost of the checkpoint primitive and by evaluating its impact across a sweep of checkpoint rates. A checkpoint is implemented as a hypercall that (i) samples the selected timing counter and (ii) appends a fixed-size record to a pre-allocated trace buffer, avoiding dynamic allocation and complex formatting in the checkpoint path. Checkpoint latency is obtained with a microbenchmark that executes N consecutive checkpoints in a tight loop on a dedicated core; we repeat the experiment and report both the average and the worst-case latency to capture run-to-run variability. Figure 4 shows the measured overhead as a function of checkpoint rate. For rates up to 10 k checkpoints/s, the performance overhead is negligible ($\leq 1\%$). At higher rates, overhead increases more rapidly; however, these configurations already provide sub-0.1 ms checkpoint granularity, which is beyond the temporal resolution required in our experiments.

5.3 Cache Coloring Auto-Tuning

Cache-coloring auto-tuning is workload dependent: the same cache partition may be acceptable for some co-runner combinations and unacceptable for others. Accordingly, the results are presented in a progressively narrower view: we first provide a workload-wide overview across all setups, then focus on a representative memory-intensive $HcVM$ workload, and finally drill down into a small set of representative $MLcVM$ co-runner scenarios.

5.3.1 Tolerance sensitivity

The auto-tuning workflow exposes a single user-defined policy parameter: the tolerance vector τ, which determines the maximum admissible degradation relative to a per-workload baseline. In this subsection, we set $\tau = (7\%, 12\%, 50\%)$ for $HcVM$, $MLcVM$, and $BEcVM$, respectively, and analyze how these values shape acceptability across the explored workload space.[2]

Figure 5 summarizes the distribution of $rel_p(c)$ across Embench co-runners for each MiBench workload and each cache-coloring configuration $c \in \{\texttt{interf}, \texttt{c1}, \ldots, \texttt{c6}\}$ (configurations detailed in Table 4). Each sample in these distributions corresponds to one consolidation instance, i.e., one MiBench workload paired with one Embench co-runner under a fixed stressor setup. The figure is organized by MiBench benchmark (rows) and dataset size

[2] Additional tolerance settings are publicly available in `https://github.com/Diogo21Costa/DAAT-MCS-results`, but are omitted here due to space constraints.

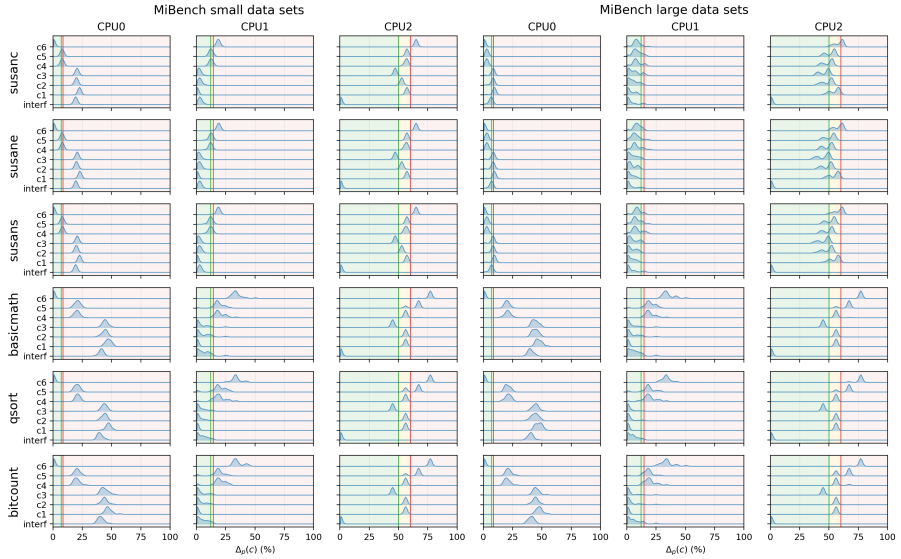

Figure 5 Workload-wide overview of $rel_p(c)$ distributions across Embench co-runners for each MiBench workload under $\tau = (7\%, 12\%, 50\%)$, where each subplot corresponds to one VM.

(columns), and reports results for $HcVM$, $MLcVM$, and $BEcVM$. In each subplot, the x-axis is $rel_p(c)$, and each line corresponds to one configuration. Configurations for which most samples lie below the tolerance threshold correspond to cases where the majority of co-runner combinations satisfy the admissible degradation, whereas configurations for which many samples lie above the threshold indicate that a large fraction of co-runners exceed the admissible degradation. These distributions therefore provide a global view of how a fixed tolerance vector shapes the set of acceptable cache-coloring choices under different workload consolidations. From these results, we observe three main takeaways.

Takeaway 1: CPU-biased allocations are not system-feasible. With no cache partitioning (*interf*), $BEcVM$, which hosts the memory-intensive stressor, uses the entire cache and achieves 100% acceptance under $\tau = (7\%, 12\%, 50\%)$, while $MLcVM$ reaches 97.73% and $HcVM$ only satisfies the tolerance in 25% of cases. At the opposite extreme, $c6$ (allocating 4 colors to $HcVM$) guarantees 100% acceptance on $HcVM$ but reduces $MLcVM$ and $BEcVM$ to 23.86% and 13.64%, respectively. Symmetrically, $c2$ (4 colors to $MLcVM$) and $c1$ (4 colors to $BEcVM$) maximize capacity for the favored VM while severely degrading $HcVM$ (13.26% for $c2$, 2.27% for $c1$). These extreme scenarios show that maximizing allocation for one core may satisfy its local tolerance but systematically compromises other workloads, confirming that per-core tuning alone cannot guarantee system-wide feasibility.

Takeaway 2: Balanced allocations expose the real trade-offs. We now examine the more balanced configurations $c3$, $c4$, and $c5$, which distribute colors more evenly. Configurations $c4$ and $c5$ (3 colors to $HcVM$) raise $HcVM$'s acceptance from 2.27% in $c1$ to 45.08–45.45%, maintain $BEcVM$ between 63.64–97.73%, but reduce $MLcVM$'s near-100% acceptance to 51.89–56.06%. Similarly, configuration $c3$ provides an intermediate allocation, achieving 22.73% on $HcVM$, 98.86% on $MLcVM$, and 100% on $BEcVM$. Even these fairer layouts exhibit residual trade-offs across VMs, showing that under $\tau = (7\%, 12\%, 50\%)$ no static partition simultaneously achieves high acceptance across all workload consolidations.

Table 5 Acceptability rate (feasible setups: $|\mathcal{C}_{\text{acc}}| > 0$) for a set of θ_{Hc} under four tolerance profiles $(\theta_{MLc}, \theta_{BE})$, aggregated over 154 consolidations (fixed $\alpha = 1.2$).

θ_{Hc} (%)	Profile A $(12, 50)$	Profile D $(15, 50)$	Profile B $(8, 50)$	Profile C $(12, 30)$
3	0.00% (0/154)	0.00% (0/154)	0.00% (0/154)	0.00% (0/154)
5	0.00% (0/154)	0.00% (0/154)	0.00% (0/154)	0.00% (0/154)
7	20.78% (32/154)	22.08% (34/154)	0.00% (0/154)	0.00% (0/154)
9	25.97% (40/154)	27.27% (42/154)	0.00% (0/154)	0.00% (0/154)
12	25.97% (40/154)	27.27% (42/154)	0.00% (0/154)	0.00% (0/154)
15	25.97% (40/154)	27.27% (42/154)	0.00% (0/154)	0.00% (0/154)
20	80.52% (124/154)	81.82% (126/154)	54.55% (84/154)	13.64% (21/154)

Table 6 Checkpoint-interval behavior for `susan-c-small` on $HcVM$, pooled across 22 co-runners.

Metric	Solo	`interf`	`c4`	`c5`	`c6`
Median interval (ms)	4.47	8.21	7.45	7.40	6.92
Std. dev. (μs)	49.1	97.0	89.4	88.8	119.1
p99 interval (ms)	4.61	8.41	7.57	7.51	6.98
Max interval (ms)	4.61	8.45	7.62	7.54	7.03

Takeaway 3: Tolerance selection controls feasibility. Cache-coloring effectiveness is fundamentally workload dependent: no single configuration achieves $\geq 90\%$ acceptance simultaneously across all VMs because each workload's sensitivity to LLC interference varies substantially under $\tau = (7\%, 12\%, 50\%)$. Only $c6$ satisfies $HcVM$'s tight 7% tolerance in all cases, while $c1$–$c3$ configurations meet the looser $MLcVM/BEcVM$ thresholds more often but systematically compromise $HcVM$'s timing constraints. DAAT-MCS exploits this structure through workload-adaptive tolerance tuning: increasing tolerance values expands the feasible configuration space by admitting configurations with larger $rel_p(c)$, whereas decreasing τ contracts the space to only those configurations yielding smaller degradations. The framework therefore exposes an explicit trade-off between configuration diversity and conservativeness, driven directly by the user-specified criticality policy.

To make this trade-off explicit, Table 5 reports the system-wide feasibility rate, i.e., the fraction of consolidations for which $\mathcal{C}_{\text{acc}} \neq \emptyset$, for different high-criticality tolerances θ_{Hc} under four representative profiles for $(\theta_{MLc}, \theta_{BE})$. Across all profiles, feasibility is zero for very strict $\theta_{Hc} \in \{3, 5\}$, indicating that none of the evaluated cache partitions can jointly satisfy the resulting degradation bounds. For profile A $(12, 50)$, feasibility becomes non-zero at $\theta_{Hc} = 7$ (20.78%) and grows modestly up to 25.97% for $\theta_{Hc} \in \{9, 12, 15\}$, before increasing sharply to 80.52% at $\theta_{Hc} = 20$. Comparing profiles highlights the role of non-critical tolerances: tightening $MLcVM$ (profile B: $(8, 50)$) and tightening $BEcVM$ (profile C: $(12, 30)$) reduce feasibility substantially, requiring a much more relaxed θ_{Hc} before any configurations become acceptable. Conversely, relaxing $MLcVM$ (profile D: $(15, 50)$) yields a small but consistent feasibility improvement over the default profile. Overall, the table confirms that tolerance selection directly controls the balance between configuration diversity (non-empty $\mathcal{C}_{\text{acc}}$ for more consolidations) and conservativeness (stricter guarantees but potentially no feasible cache partition) under the explored candidate set $\mathcal{C}$.

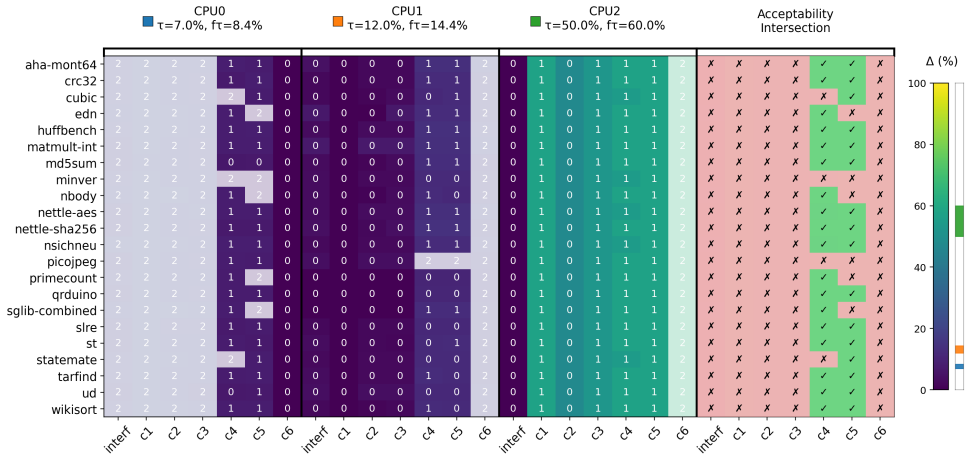

Figure 6 Full heatmap for `susan-c-small`: per-co-runner degradation $rel_p(c)$ and acceptance class for each CPU and cache-color configuration.

5.3.2 Memory-Intensive Workload Use Case

Next, we narrow the scope to a single representative high-criticality workload on $HcVM$, chosen from the MiBench Automotive suite: `susan-c-small`. This benchmark is selected because it is memory intensive and consistently exposes the key tuning tension between local isolation of $HcVM$ and system-wide feasibility once $MLcVM$ and $BEcVM$ tolerances are enforced. Figure 6 shows a heatmap for `susan-c-small` across all 22 Embench co-runners. Rows correspond to Embench benchmarks, while columns are grouped by CPU; within each CPU group, columns enumerate `interf` and `c1`–`c6` in increasing order of colors assigned to $HcVM$. Each cell reports the observed degradation $rel_p(c)$ (color scale) and its acceptance class (numeric value) under the same tolerance policy as in the previous subsection.

On $HcVM$, `susan-c-small` is evaluated with the tightest tolerance (7% for class 0 and 8.4% for class 1). Under this setting, `interf`, `c1`, `c2`, and `c3` (no coloring or only 2 colors assigned to $HcVM$) never satisfy the tolerance, and all setups are classified as class 2. Configurations `c4` and `c5` (3 colors assigned to $HcVM$) start to admit feasible setups: `c4` is tolerance-compliant in 19 out of 22 setups, while `c5` is compliant in 17 setups. Finally, `c6` (4 colors assigned to $HcVM$) fully isolates the critical workload, yielding class 0 in all 22 setups.

However, improving isolation on $HcVM$ induces degradation on the remaining VMs. For $MLcVM$ and $BEcVM$, `c6` systematically compromises their tolerance thresholds (12% and 50%, respectively): all `c6` setups are classified as class 2. Therefore, `c6` is rejected not because it fails on the critical VM, but because the system-wide intersection step removes it once $MLcVM$ and $BEcVM$ constraints are enforced. Under the same policy, the tuner restricts the deployable set primarily to `c4` and `c5`, which preserve feasibility across most consolidations.

Tail behavior on the critical VM. We report the p99 and maximum checkpoint-interval duration for $HcVM$ to capture bursty interference effects that are not visible in aggregate execution time and to assess whether system-feasible cache partitions (`c4`/`c5`) also improve worst-case behavior. While these measurements do not constitute a formal WCET guarantee, they provide an empirical predictability indicator by quantifying how cache partitioning

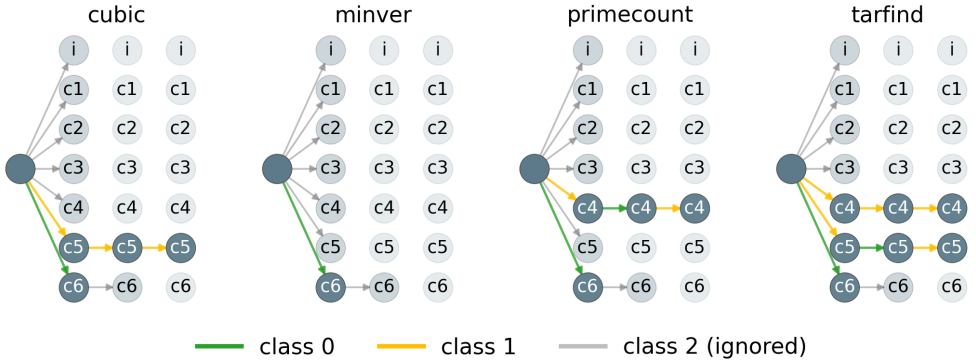

Figure 7 Representative co-runner scenarios for `susan-c-small`: decision trace of configuration feasibility across $HcVM \rightarrow MLcVM \rightarrow BEcVM$ under the intersection-based acceptability policy.

shapes both high-percentile and worst observed timing behavior under contention. As shown in Table 6, with no cache partitioning (*interf*), both the p99 and the maximum checkpoint interval nearly double relative to the solo scenario, increasing worst-case execution from 4.6 ms to over 8.4 ms. Cache partitioning significantly mitigates this effect: $c4$ and $c5$, which are system-feasible across most consolidations, reduce the p99 increase to 64% and 63%, respectively, recovering a substantial fraction of the tail predictability lost under interference. Although $c6$ achieves the lowest median, p99, and maximum interval times on $HcVM$, it also exhibits the largest observed run-to-run variability, with a standard deviation of 119 μs, exceeding even the jitter observed under unmanaged interference. Combined with the fact that $c6$ systematically compromises tolerance constraints on $MLcVM$ and $BEcVM$, these results confirm that $c6$ is suboptimal from a system-level predictability standpoint. Overall, $c4$ and $c5$ strike a more balanced trade-off: they substantially reduce tail latency relative to *interf* while maintaining lower variability and preserving system-wide feasibility.

Comparing `c4` and `c5` on $MLcVM$, both configurations yield 21 acceptable setups, so $MLcVM$ alone does not discriminate between them. On $BEcVM$, all 22 setups under both `c4` and `c5` are acceptable (class ≤ 1), providing no additional restriction. In this use case, the auto-tuner therefore selects between `c4` and `c5` using the secondary ranking criteria from Section 4, while rejecting `c6` even though it is ideal for $HcVM$ alone. Overall, across the 22 Embench co-runners, the auto-tuner identifies suitable colorings as follows: `c4` and `c5` both acceptable in 14 setups, `c4` only in 4 setups, `c5` only in 2 setups, and no feasible configurations in 2 setups. To make this pruning process explicit, Figure 7 depicts decision traces for four representative co-runners. Each tree applies acceptability constraints in sequence ($HcVM \rightarrow MLcVM \rightarrow BEcVM$), showing how the initial candidate set is reduced as additional per-CPU constraints are enforced.

No feasible configuration *(minver)*. Only `c6` satisfies $HcVM$'s tolerance; however, `c6` does not satisfy $MLcVM$'s tolerance, yielding an empty feasible set. For this setup, DAAT-MCS reports that no cache-coloring configuration satisfies the specified tolerances, implying that deployment requires relaxing τ and/or rejecting this consolidation.

`c4` only *(primecount)*. Both `c4` and `c6` satisfy $HcVM$'s tolerance. After enforcing $MLcVM$ and $BEcVM$ constraints, `c6` is excluded because it does not satisfy the tolerance on at least one non-critical CPU, while `c4` remains acceptable across all CPUs. Hence, `c4` is the only feasible configuration for this consolidation.

`c5` only *(cubic)*. Both `c5` and `c6` satisfy $HcVM$'s tolerance. Enforcing the $MLcVM$ constraint excludes `c6` because it does not satisfy the tolerance, while `c5` remains acceptable and also satisfies the $BEcVM$ constraint, making `c5` the only feasible configuration.

`c4` and `c5` feasible *(tarfind)*. Multiple configurations satisfy $HcVM$'s tolerance, but enforcing $MLcVM$ and $BEcVM$ constraints reduces the candidate set to $\{\texttt{c4}, \texttt{c5}\}$. In this case, both `c4` and `c5` satisfy system-wide tolerances, and the final selection is delegated to the secondary ranking criteria defined in Section 4.

6 Discussion

In this paper, DAAT-MCS is presented primarily for LLC cache coloring and evaluated using the Bao Hypervisor [27] as a concrete use case; nevertheless, the workflow does not rely on Bao-specific design choices. Aditionally, the current profiling infrastructure introduces practical overheads that can be removed with short- to mid-term engineering refinements. This section discusses these limitations and outlines how the presented artifacts can be extended and deployed in broader settings.

Hypervisor dependence and portability. Our evaluation targets a specific SPH; however, DAAT-MCS is not tied to Bao's internal design and can be adapted to other SPHs (e.g., Jailhouse). In practice, porting DAAT-MCS to a different SPH requires only two integration points: (i) implementing the guest-to-monitor instrumentation interface used by the profiler (i.e., the hypercall handlers for session start/stop and checkpoint setup, as described in Section 3); and (ii) extending the configuration back-end to generate the corresponding hypervisor configuration artifacts (e.g., VM descriptions and cache-partition assignments in the target SPH's format, when using an hypervisor). More broadly, the same measurement-and-selection pipeline can be deployed without a hypervisor by integrating instrumentation and enforcement into the OS kernel; for example, prior work such as MemGuard [49] follows this paradigm for memory-bandwidth reservation within the Linux kernel.

Extending interference mitigation techniques. The current work focuses primarily on LLC cache coloring (altough the framework also supports memory-bandwidth regulation), but modern high-end embedded platforms increasingly expose additional mechanisms (e.g., Arm QoS regulators and standardized partitioning/monitoring interfaces such as Arm MPAM). These mechanisms are complementary to cache coloring because they can bound contention that manifests beyond the LLC [51], which cache partitioning per se cannot directly regulate. Crucially, extending DAAT-MCS to such mechanisms does not require changing the core auto-tuning logic (tolerance classification plus system-wide feasibility intersection); instead, it requires extending the configuration space definition and generation pipeline so that (i) the new mechanism parameters are described in the framework's input specification, (ii) candidate configurations enumerating those parameters are generated, and (iii) those candidates are compiled/deployed by the corresponding back-end (hypervisor configuration generation when using an SPH, or kernel configuration hooks when operating at OS level).

Profiler footprint and dedicated-core overhead. The current implementation dedicates one CPU core to a Linux-based control VM that coordinates profiling, reducing the number of cores available for workload consolidation from 4 to 3 on the quad-core Xilinx UltraScale+ ZCU104. In addition, the Linux VM increases the profiling image size due to the guest kernel

and userspace tooling. These trade-offs can be mitigated through two orthogonal directions: (i) offloading profiling coordination to auxiliary platform components (e.g., FPGA fabric or an auxiliary cluster when available), removing persistent CPU dedication while preserving hypervisor mediation; or (ii) replacing the always-on control VM with a minimal "profile-then-run" workflow in which the profiler executes briefly at the beginning and end of the experiment to synchronize, collect, and export traces, requiring only two hypervisor transitions per profiling cycle while preserving full core availability for production deployments.

Future Work. Short-term extensions will address current limitations by (i) offloading profiling coordination to external platform resources (e.g., RPU and/or FPGA fabric) to avoid dedicating a CPU core, (ii) supporting VM stacking with minimal context switches to further reduce trace-collection overhead, and (iii) integrating additional interference mitigation mechanisms (e.g., QoS regulators) into the same tolerance-guided workflow. Future work also encompasses the improvement of performance characterization by complementing (or replacing) user-defined checkpoints with PMU-based monitoring (e.g., cache misses and memory stalls), providing richer workload information for tolerance classification.

7 Related Work

Over the last decade, interference in MCSs has been explored along two main directions: (i) characterizing and exposing sources of interference across shared resources, and (ii) developing and assessing mitigation techniques that improve temporal isolation. Both directions have been strongly supported by state-of-the-art benchmark suites, leveraging application-oriented workloads (e.g., MiBench [18], SD-VBS [47], TACLeBench [15]) and synthetic workloads (e.g., lmbench [29]), which enable reproducible stress and evaluation across platforms. However, as modern workloads and platforms become more complex, isolated benchmark-driven studies are often insufficient to uncover end-to-end interference paths and to assess mitigation effectiveness on real hardware. This has motivated a complementary body of work on frameworks and tools that systematize interference generation, measurement, and analysis, alongside continued advances in mitigation mechanisms and in hardware support for partitioning and regulation.

Frameworks and tooling for interference analysis. Several frameworks and tools have been proposed to systematically create interference scenarios, measure slowdowns, and support integration-time decision making on real hardware. RT-Bench provides an extensible benchmark framework for real-time experimentation by attaching task semantics (e.g., periods and deadlines) to co-scheduled workloads [32]. FrATM2 automates microbenchmark generation under diverse contention scenarios and learns memory-subsystem timing models that can be leveraged by response-time analysis [43]. SP-IMPact provides an open-source workflow to deploy VM configurations on static-partitioning hypervisors and assess the impact of mitigation techniques such as cache coloring and memory-bandwidth reservation under consolidation [13]. At the tooling level, RapiDaemons and MinervaSys Architect provide stress/inference generation and inspection capabilities to expose interference paths and assess mitigation impact [46, 45]. Beyond hypervisor-centered workflows, Dasari et al. propose a framework to compute memory-contention bounds under shared-bus arbitration assumptions [14], and Oliveira et al. provide an empirical framework to quantify contention on low-end multi-core microcontrollers [33]. In the cache domain, Mancuso et al. present a real-time cache-management framework that uses page coloring to control cache-set usage and reason about timing impact [26]. Finally, MEMSCOPE provides an open-source, kernel-level

framework to characterize heterogeneous memory subsystems under controlled contention, enabling precise measurement of latency/bandwidth behavior and interference effects at the OS level [16].

Interference mitigation and configuration. A second line of work focuses on mitigation mechanisms and on configuration/allocation strategies that bound interference under mixed-criticality constraints. In the cache domain, works study cache partitioning, cache-aware virtualization, and cache-centric isolation (e.g., page-coloring-based management and cache DoS prevention) [26, 6]. In the memory domain, extensive research addresses bandwidth regulation and memory-system control across different enforcement models: OS-level regulators (e.g., MemGuard and subsequent bandwidth-management systems), external/polling-based policing (e.g., MemPol), and studies of regulation dynamics and platform-specific constraints (e.g., DynamIQ/DSU-oriented regulators) [49, 50, 52, 2, 41, 36]. Complementary approaches target specific bottlenecks and heterogeneity, including QoS-enabled accelerator/CPU bandwidth management, programmable-logic-assisted memory scheduling infrastructures, and hardware support for contention tracking in shared caches [42, 20, 5]. Finally, prediction and virtualization-oriented techniques complement pure regulation by either anticipating interference at runtime or strengthening isolation within virtualized deployments [40, 24, 31].

Gap analysis. Prior work has made substantial progress on (i) frameworks and tools to generate and measure interference under consolidation, and (ii) analytical or learned models that explain or predict contention-induced slowdowns. DAAT-MCS targets a different point in the design space: it is not intended to provide new insight into interference mechanisms, nor to propose a new interference-mitigation primitive. Instead, it fills a practical gap that remains largely manual in current integration workflows: the automatic configuration of existing mitigation techniques for concrete mixed-criticality deployments under explicit, per-core degradation requirements. Concretely, given a finite candidate configuration set $\mathcal{C}$, DAAT-MCS provides an end-to-end workflow that (i) profiles each candidate on real hardware via a hypervisor-integrated checkpoint tracer and (ii) applies tolerance-guided classification plus constraint reasoning to derive $\mathcal{C}_{\text{acc}}$, the set of system-wide feasible configurations. Compared to modeling/prediction approaches, DAAT-MCS does not seek a reusable interference model; it makes feasibility decisions directly from measured traces, thereby retaining platform- and workload-specific effects. Compared to evaluation frameworks that primarily report performance/isolation outcomes across configurations, DAAT-MCS provides a deterministic selection procedure that maps measurements into deployable configurations aligned with certification-oriented tolerance bounds, replacing ad-hoc trial-and-error with a repeatable offline step. To the best of our knowledge, no prior work provides this specific combination of hypervisor-level checkpoint profiling, tolerance-guided classification, and system-wide feasibility intersection to automatically configure interference mitigation techniques for static-partitioning mixed-criticality consolidation.

8 Conclusion

This work introduced DAAT-MCS, an offline framework that automates LLC cache-color partition selection for mixed-criticality consolidation by combining hypervisor-level checkpoint profiling with tolerance-guided classification and system-wide feasibility reasoning. Rather than relying on ad-hoc, platform-specific trial-and-error, DAAT-MCS selects system-wide acceptable cache partitions from a finite candidate space by enforcing explicit per-core

degradation tolerances. We evaluated DAAT-MCS on an ARMv8-A Xilinx UltraScale+ ZCU104 platform across 264 workload combinations and 7 cache-partitioning configurations (1848 setups), collecting more than 60 hours of automated profiling traces. All artifacts are publicly released to support reproducible research and follow-on work.

References

1 Jaume Abella, Carles Hernandez, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Perez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. *10th IEEE International Symposium on Industrial Embedded Systems*, pages 1–10, 2015.

2 Ankit Agrawal, Renato Mancuso, Rodolfo Pellizzoni, and Gerhard Fohler. Analysis of Dynamic Memory Bandwidth Regulation in Multi-core Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 230–241, 2018. doi:10.1109/RTSS.2018.00040.

3 Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd Annual ACM Southeast Conference*, pages 267–272, 2004. doi:10.1145/986537.986601.

4 Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I. Davis. Evaluation of Cache Partitioning for Hard Real-Time Systems. In *26th Euromicro Conference on Real-Time Systems*, pages 15–26, 2014.

5 Javier Barrera, Leonidas Kosmidis, Hamid Tabani, Jaume Abella, and Francisco J. Cazorla. Hardware support for contention tracking in CPU and GPU last-level cache. *Journal of Systems Architecture*, 169:103591, 2025. doi:10.1016/J.SYSARC.2025.103591.

6 Michael Bechtel and Heechul Yun. Denial-of-Service Attacks on Shared Cache in Multicore: Analysis and Prevention. *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 357–367, 2019.

7 Francisco J. Cazorla, Leonidas Kosmidis, Enrico Mezzetti, Carles Hernandez, Jaume Abella, and Tullio Vardanega. Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM CSUR*, 52(1), 2019. doi:10.1145/3301283.

8 Jon Perez Cerrolaza, Roman Obermaisser, Jaume Abella, Francisco J. Cazorla, Kim Grüttner, Irune Agirre, Hamidreza Ahmadian, and Imanol Allende. Multi-core devices for safety-critical systems: A survey. *ACM CSUR*, 53(4), 2020.

9 Diogo Costa. DAAT-MCS. Software, version 1.0. (visited on 2026-06-16). URL: https://github.com/Diogo21Costa/DAAT-MCS, doi:10.4230/artifacts.26724.

10 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, José Martins, Fabrizio Tronci, Alessandro Biasci, and Sandro Pinto. IRQ Coloring: Mitigating Interrupt-Generated Interference on ARM Multicore Platforms. *Fourth Workshop on Next Generation Real-Time Embedded Systems*, 108:2:1–2:13, 2023. doi:10.4230/OASIcs.NG-RES.2023.2.

11 Diogo Costa, Luca Cuomo, Daniel Oliveira, Ida Maria Savino, Bruno Morelli, José Martins, Alessandro Biasci, and Sandro Pinto. IRQ Coloring and the Subtle Art of Mitigating Interrupt-Generated Interference. *IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–56, 2023.

12 Diogo Costa, Jose Martins, and Sandro Pinto. Beyond the Bermuda Triangle of Contention: IOMMU Interference in Mixed Criticality Systems. In *IEEE 31st International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 183–194, August 2025.

13 Diogo Costa, Gonçalo Moreira, Afonso Oliveira, José Martins, and Sandro Pinto. SP-IMPact: A Framework for Static Partitioning Interference Mitigation and Performance Analysis. In Patrick Meumeu Yomsi and Stefan Wildermann, editors, *Sixth Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2025)*, volume 128 of *Open Access Series in Informatics (OASIcs)*, pages 5:1–5:15, 2025. doi:10.4230/OASIcs.NG-RES.2025.5.

14 Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Syst.*, 52(3):272–322, May 2016. doi:10.1007/S11241-015-9229-9.

15 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Bjørn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, {Rasmus Bo} Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. *Open Access Series in Informatics*, 55:2.1–2.10, 2016.

16 Golsana Ghaemi, Gabriel Franco, Kazem Taram, and Renato Mancuso. MEMSCOPE: Open-Source Kernel-Level Framework for Heterogeneous Memory Characterization. In *IEEE Real-Time Systems Symposium*, pages 500–513, 2025. doi:10.1109/RTSS66672.2025.00047.

17 Giovani Gracioli, Rohan Tabish, Renato Mancuso, Reza Mirosanlou, Rodolfo Pellizzoni, and Marco Caccamo. Designing Mixed Criticality Applications on Modern Heterogeneous MPSoC Platforms. *31st Euromicro Conference on Real-Time Systems*, 133:27:1–27:25, 2019. doi:10.4230/LIPIcs.ECRTS.2019.27.

18 M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14, 2001.

19 Thomas A. Henzinger and Joseph Sifakis. "The Embedded Systems Design Challenge". *FM 2006: Formal Methods*, pages 1–15, 2006. doi:10.1007/11813040_1.

20 Denis Hoornaert, Shahin Roozkhosh, and Renato Mancuso. A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, volume 196 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:22, 2021. doi:10.4230/LIPIcs.ECRTS.2021.2.

21 C.J. Janraj, T. Venkata Kalyan, Tripti Warrier, and Madhu Mutyam. Way Sharing Set Associative Cache Architecture. In *25th International Conference on VLSI Design*, pages 251–256, 2012. doi:10.1109/VLSID.2012.79.

22 Sophie Kahlen and Jan Reineke. A Unified Framework for Quantitative Cache Analysis. In *IEEE 31st Real-Time and Embedded Technology and Applications Symposium*, pages 54–67, May 2025. doi:10.1109/RTAS65571.2025.00018.

23 Hyoseung Kim and Ragunathan (Raj) Rajkumar. Real-time cache management for multi-core virtualization. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16, 2016.

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

25 Tamara Lugo, Santiago Lozano, Javier Fernández, and Jesus Carretero. A Survey of Techniques for Reducing Interference in Real-Time Applications on Multicore Platforms. *IEEE Access*, 10:21853–21882, 2022. doi:10.1109/ACCESS.2022.3151891.

26 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. *IEEE 19th Real-Time and Embedded Technology and Applications Symposium*, pages 45–54, 2013. doi:10.1109/RTAS.2013.6531078.

27 José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems. *Workshop on Next Generation Real-Time Embedded Systems (NG-RES)*, 77:3:1–3:14, 2020. doi:10.4230/OASIcs.NG-RES.2020.3.

28 José Martins and Sandro Pinto. Shedding Light on Static Partitioning Hypervisors for Arm-based Mixed-Criticality Systems. *IEEE 29th Real-Time and Embedded Technology and Applications Symposium*, pages 40–53, 2023.

29 Larry W McVoy, Carl Staelin, et al. Lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

30 Sparsh Mittal. A Survey of Techniques for Cache Locking. *ACM Trans. Des. Autom. Electron. Syst.*, 21(3), May 2016. doi:10.1145/2858792.

31 Paolo Modica, Alessandro Biondi, Giorgio Buttazzo, and Anup Patel. Supporting temporal and spatial isolation in a hypervisor for ARM multicore platforms. In *IEEE International Conference on Industrial Technology (ICIT)*, pages 1651–1657, 2018. doi:10.1109/ICIT.2018.8352429.

32 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 '22, pages 184–195, 2022. doi:10.1145/3534879.3534888.

33 Daniel Oliveira, Weifan Chen, Sandro Pinto, and Renato Mancuso. Investigating and Mitigating Contention on Low-End Multi-Core Microcontrollers. In *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023*, CPS-IoT Week '23, pages 221–226, 2023. doi:10.1145/3576914.3587513.

34 Rob Palin, David Ward, Ibrahim Habli, and Roger Rivett. ISO 26262 safety cases: Compliance and assurance. In *6th IET International Conference on System Safety 2011*, page B12. IET, 2011.

35 Sandro Pinto, Jorge Pereira, Tiago Gomes, Adriano Tavares, and Jorge Cabral. LTZVisor: TrustZone is the Key. *29th Euromicro Conference on Real-Time Systems*, 76:4:1–4:22, 2017. doi:10.4230/LIPIcs.ECRTS.2017.4.

36 Ashutosh Pradhan, Daniele Ottaviano, Yi Jiang, Haozheng Huang, Jiajia Zhang, Alexander Zuepke, Andrea Bastoni, and Marco Caccamo. Predictable Memory Bandwidth Regulation for DynamIQ Arm Systems. In *IEEE 31st International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 126–137, 2025. doi:10.1109/RTCSA66114.2025.00022.

37 I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 114–123, 2002.

38 Jon Altonaga Puente, Enrico Mezzetti, Irune Agirre, Jaume Abella, and Francisco Javier Cazorla-Almeida. ROSGuard: A Bandwidth Regulation Mechanism for ROS2-based Applications. *ArXiv*, abs/2506.04640, 2025. doi:10.48550/arXiv.2506.04640.

39 Ralf Ramsauer, Jan Kiszka, Daniel Lohmann, and Wolfgang Mauerer. Look Mum, no VM Exits! (Almost), 2017.

40 Ahsan Saeed, Daniel Mueller-Gritschneder, Falk Rehm, Arne Hamann, Dirk Ziegenbein, Ulf Schlichtmann, and Andreas Gerstlauer. Learning based Memory Interference Prediction for Co-running Applications on Multi-Cores. In *ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2021. doi:10.1109/MLCAD52597.2021.9531245.

41 Eric Seals, Michael Bechtel, and Heechul Yun. BandWatch: A System-Wide Memory Bandwidth Regulation System for Heterogeneous Multicore. In *IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 38–46, September 2023.

42 Parul Sohal, Rohan Tabish, Ulrich Drepper, and Renato Mancuso. Profile-driven memory bandwidth management for accelerators and CPUs in QoS-enabled platforms. *Real-Time Syst.*, 58(3):235–274, September 2022. doi:10.1007/S11241-022-09382-X.

43 Andrea Stevanato, Matteo Zini, Alessandro Biondi, Bruno Morelli, and Alessandro Biasci. Learning Memory-Contention Timing Models With Automated Platform Profiling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(11):3816–3827, 2024. doi:10.1109/TCAD.2024.3449237.

44 Connor Sullivan, Alex Manley, Mohammad Alian, and Heechul Yun. Per-Bank Bandwidth Regulation of Shared Last-Level Cache for Real-Time Systems. In *IEEE Real-Time Systems Symposium*, pages 336–348, 2024. doi:10.1109/RTSS62706.2024.00036.

45 Minerva Systems. Optimize Safe and Secure Integration on Multiprocessor Systems on Chip. White Paper, 2023. Accessed: 2026-02-20.

46 Steven H. VanderLeest and Samuel R. Thompson. Measuring the Impact of Interference Channels on Multicore Avionics. In *AIAA/IEEE 39th Digital Avionics Systems Conference (DASC)*, pages 1–8, 2020.

47 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 *IEEE International Symposium on Workload Characterization (IISWC)*, pages 55–64. IEEE, 2009. doi:10.1109/IISWC.2009.5306794.

48 Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: a dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 381–392, 2014. doi:10.1145/2628071.2628104.

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

50 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016. doi:10.1109/TC.2015.2425889.

51 Matteo Zini, Giorgiomaria Cicero, Daniel Casini, and Alessandro Biondi. Profiling and controlling I/O-related memory contention in COTS heterogeneous platforms. *Software: Practice and Experience*, 52(5):1095–1113, 2022. doi:10.1002/SPE.3053.

52 Alexander Zuepke, Andrea Bastoni, Weifan Chen, Marco Caccamo, and Renato Mancuso. MemPol: polling-based microsecond-scale per-core memory bandwidth regulation. *Real-Time Systems*, pages 1–44, 2024.