Interferences within a certifiable design methodology for high-performance multi-core platforms

Mohamed Amine KHELASSI[1], Felix SUCHERT[3], Abderaouf AMALOU[2], Benjamin LESAGE[4], Anika CHRISTMANN[5], Robin HAPKA[5], Jeronimo CASTRILLON[3], Mihail ASAVOAE[1], Mathieu JAN[1], Claire PAGETTI[4], Selma SAIDI[5]

[1] *Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France*

[2] *Nantes Université, École Centrale Nantes, LS2N, CNRS UMR 6004, F-44000 Nantes, France*

[3] *Technische Universität Dresden, Dresden, Germany*

[4] *ONERA, Toulouse, France*

[5] *Technische Universität Braunschweig, Germany*

Abstract

The adoption of high-performance multi-core platforms in avionics and automotive systems introduces significant challenges in ensuring predictable execution, primarily due to shared resource interferences. Many existing approaches study interference from a single angle—for example, through hardware-level analysis or by monitoring software execution. However, no single abstraction level is sufficient on its own. Hardware behavior, program structure, and system configuration all interact, and a complete view is needed to understand where interferences come from and how to reduce them. In this paper, we present a methodology that brings together several tools that operate at different abstraction levels. At the lowest level, PHYLOG provides a formal model of the hardware and identifies possible interference channels using micro-architectural transactions. At the program level, machine learning analysis locates the exact parts of the code that are most sensitive to shared-resource contention. At the compilation level, MLIR-based transformations use this information to reshape memory access patterns and reduce pressure on shared resources. Finally, at the system level, Linux cgroups enforce static execution constraints to prevent highly interfering tasks from running together. The goal of our approach is to reduce memory interference and improve the system's predictability, thereby easing the certification process of multi-core systems in safety-critical domains.

Keywords Memory interferences, MLIR framework, Machine learning, Linux cgroups

1. Introduction

The increasing integration of high-performance multi-core platforms in safety-critical domains such as avionics and automotive systems presents both opportunities and challenges. These platforms offer the computational capabilities required for modern workloads but make it difficult to meet strict timing and safety requirements imposed by standards like ISO 26262 and DO-178C [15, 9, 28]. A key source of difficulty lies in the complex and often unpredictable interactions between concurrent applications (or tasks within an application) competing for shared hardware resources. These interactions lead to hardly predictable delays, that are called *interferences*.

Understanding and controlling inter-core interferences is critical to ensure predictable system behavior. However, such interferences are difficult to model, analyze, and mitigate due to their dependency on hardware architecture, executive layer (e.g. operating system or hypervisor) and execution patterns. Without a structured methodology, this unpredictability complicates certification efforts and increases system design costs.

There has been extensive research to identify and analyze interferences. However, each approach focuses on individual aspects such as timing anomaly analysis or *micro-architectural* interference identification. By *micro-architectural*, we mean abstracting the application and executive layer as a set of micro-transactions (e.g. core reading a data in a DDR bank). Such methods, although valuable, do not provide the comprehensive view required to accurately model and mitigate interference across the complex interactions between software execution and hardware behavior. Therefore, there exists a clear demand for an integrated methodology that bridges hardware and software analyses, enabling more accurate interference modeling and mitigation.

In this paper, we propose a combined hardware/software methodology for modeling, analyzing, and reducing interferences on multi-core systems. Our approach integrates several layers (see Figure 1). We identify *micro-architectural* interferences with PHYLOG approach, that relies on a platform model and targeted micro-benchmarks. We use machine learning to find code regions sensitive to those interferences and annotate the application code accordingly. Thanks to this information, MLIR compiler optimizations are applied in order to reshape mem-

ory access to reduce contention and we configure Linux cgroups to control how resources are shared between tasks.

The remainder of this paper is structured as follows. The next Section 2 details our methodology. Section 4 presents our potential use cases. Finally, the conclusion 6 outlines the directions for future research.

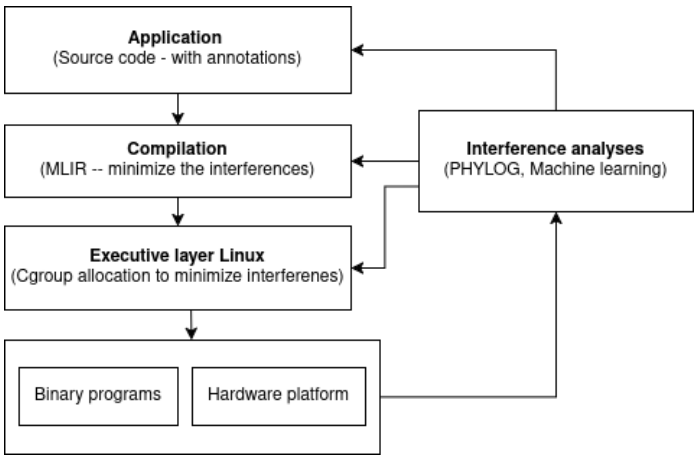

Figure 1. Overview of the interference-aware design methodology.

2. Methodology

We aim to present an end-to-end methodology for detecting, quantifying, and reducing memory interference in multicore systems. To address memory interference in a practical way, we propose a flow that combines analysis, transformation, and runtime mitigation. The intuition is to integrate compile-time and runtime strategies guided by PHYLOG and machine learning analysis to both understand and control timing variability. To address the challenge, our methodology uses three complementary techniques: (1) interference analysis using the PHYLOG framework and machine learning, (2) compilation-level optimizations based on MLIR, and (3) static resource allocation using Linux cgroups.

2.1. Micro-architectural Modeling

The PHYLOG [7] methodology was designed to assist applicants in certifying their multi-core processors in the aeronautical domain. A platform description, in the PHYLOG modeling language (PML), captures the knowledge about the characteristics of the platform based on the available documents and the applicant's assessments. It also captures the target configuration, including hardware and software settings such as the mapping of applications hosted on the platform to cores.

The PML model serves as the backbone for the identification of *interference channels* through interference calculus. Interference channels, as per AMC20-193 [14], are hardware resources whose use by an application might cause variations in its functional or temporal behaviour over its behaviour in isolation. Interference calculus thus provides transactions combinations expected to be free from (or to be suffering from) interference. Each transaction captures a path an application may exercise through hardware resources.

The interfering or interference-free combinations then support the validation of the model, understanding latent interference sources, and the quantification of said interference where applicable. This is achieved through micro-benchmarks each exercising individual transactions. Combinations of micro-benchmarks [1, 12], matching the combinations of interest identified through interference calculus, should confirm the verdict of the analysis. Otherwise, refinements to the model are required.

The application and executive layer are abstracted as a set of micro-architectural transactions. The identified interferences are also expressed as microarchitectural interferences. Linking this knowledge to application code and behaviour is far from simple. This is the reason why it is then combined with a complementary analysis explained below.

2.2. Interference Modeling using Machine Learning

We build on the microbenchmarking approach proposed by Courtaud et al [12] to analyze how multi-core memory interferences affect code execution performance. Specifically, we gather some metrics from the Performance Monitoring Unit (PMU) and combine it with observed contention metrics (defined as the ratio between the execution time under contention and in isolation) we formulate an inverse machine learning problem [4]. Our goal is to identify precisely which segments of the code are most vulnerable to performance degradation when competing for shared resources such as L2 caches, memory buses, and DRAM utilization. This approach offers a new way to pinpoint the code regions most sensitive to resource contention and create what we will call a code heatmap artefact that can be used as cost function for MLIR optimization pass and as input data for the cgroup allocation heuristic.

2.3. Interference reduction via compilation

This part of the methodology focuses on reducing interference through compile-time and run-time techniques applied to parallel dataflow applications. Traditional compiler optimizations improve cache locality for single-threaded code, but they do not address the interference caused by parallel execution, especially when many dataflow nodes compete for a limited number of hardware threads. This leads to frequent task switching and extra cache misses.

To handle this, we first analyze the dataflow graph of the application and fuse compatible nodes to reduce the overall number of parallel tasks. This is done using an MLIR-based compilation flow (Etna), which transforms the application into a dedicated dataflow IR and applies fusion when nodes operate at the same rate and communicate unconditionally. Fewer tasks mean less oversubscription, fewer context switches, and lower memory interference.

At runtime, we complement this with the HARP resource manager, which monitors performance metrics and adjusts thread allocation. By prioritizing nodes that suffer more cache misses, HARP helps limit interference during execution without requiring changes to user code.

Together, these compile-time graph transformations and runtime resource adjustments reduce memory contention in parallel applications and improve execution predictability.

2.4. Interference reduction via cgroups

Cgroups provide a way to partition hardware resources in Linux, making it possible to restrict how tasks share CPU cores, memory, and other components. In our methodology, cgroups are used at runtime to prevent tasks that strongly interfere with each other from running at the same time.

We rely on an interference matrix, built from PHYLOG analysis and machine learning results, which quantifies how much slowdown each pair of tasks causes when they run together. Based on this matrix, we build a static cgroup hierarchy in which tasks that show high mutual interference are separated into different cgroups and controlled using the freezer subsystem, ensuring they do not execute concurrently. Tasks that do not interfere significantly can be grouped together.

This static cgroup configuration helps enforce predictable behaviour by isolating interfering tasks at the operating-system level, complementing the compile-time and program-level optimizations used in other parts of the methodology.

3. How things interact

Our methodology integrates distinct yet complementary layers to reduce memory interference in interference-aware design flow.

- PHYLOG performs a platform-level analysis based on a formal model of the hardware and software configuration. It identifies potential interference channels using micro-architectural transactions and interference calculus.
- Machine learning analysis processes data from targeted microbenchmarks and performance counters. It pinpoints which code segments are most sensitive to interference, producing a code heatmap that characterizes application-level interference behaviour.
- MLIR-based transformations use this heatmap as a cost model to reshape memory access behaviour
- The cgroup configuration heuristic takes as input both the PHYLOG interference channels and the machine learning-based heatmap. It uses this joint interference characterization to assign tasks to cgroups and tune resource limits.

4. Use-Cases & Evaluation

In our evaluation, we focus exclusively on the Raspberry Pi 4 platform, which features four Cortex–A72 cores with private 32 KB L1 data caches and a shared 1 MB L2 cache. The system runs Raspbian, providing a lightweight environment suitable for rapid instrumentation. As the test workload, we use a parameterized matrix multiplication kernel, where the sizes N, M, and K control the dimensions of matrices A and B. This setup allows us to study how variations in memory-access patterns affect interference sensitivity and prediction accuracy.

```
#define N 256
#define K 4096
#define M 9

int main() {
    int A[N][K];
    int B[K][M];
    Initialize_Matrix_Random(N, K, A, 1, 10);
    Initialize_Matrix_Random(K, M, B, 1, 10);
    int C[N][M];
    MM(A,B,C);
    return 0;
}
```

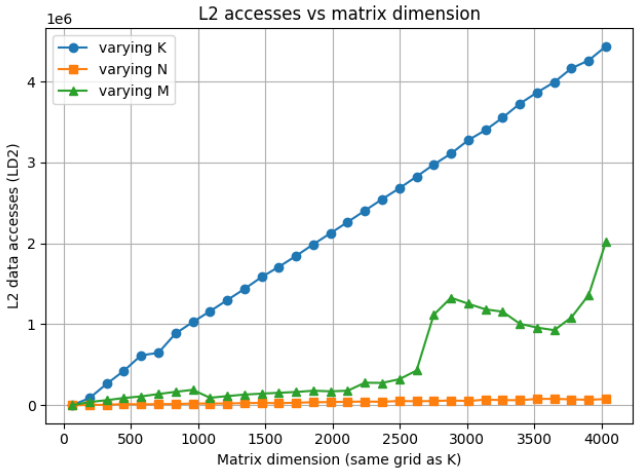

Figure 2. Sensitivity of L2 Accesses to Variations in Matrix Dimensions (M, N and K) starting from 64 with a step of 128, the other dimensions are fixed into 32.

To study the sensitivity to interferences we measured the number of L2-cache accesses by fixing one of the parameters M, N, or K to 32 while varying the remaining parameter. The varying parameter started at 64 and increased with a step of 128 until it exceeded 4096. The resulting curve, shown in the figure, indicates that matrix multiplications with a large value of K—corresponding to irregular matrix shapes—produce significantly more L2-cache accesses. Although such large K values may appear unusual, they are common in modern inference workloads, particularly in Transformer-based models and LLMs, where many matrix operations involve highly non-square shapes (with K much larger than M and N). Moreover, a higher number of L2 accesses is advantageous for studying interference effects.

4.1. Micro-architectural Modeling

To support the identification of potential interference channels on the target Raspberry Pi 4 platform, we built a model of the board. Figure 3 presents an overview of the identified hardware resources. Hardware resources have been classified (and color-coded) following the PML nomenclature into Initiators, Transporters, or Targets for transactions. The model was built through a first review of the available documentation on the board [27], the embedded Broadcom BCM2711 System-on-Chip (SoC) [26], and its ARM A72 processor [25].

The Raspberry Pi 4 board comprises a Broadcom BCM2711 SoC, configurable LPDDR4 SDRAM, and a number of supporting interfaces such as USB, GPIO, HDMI, etc. Our efforts focused on capturing the resources relevant to considered applicative use case; the model only captures interfaces and peripherals at the board- and SoC-level with a high granularity. We consider a conservative model, exemplified by the integrated VideoCore GPU (*iGPU*), where each such device is a composite with a single target and a single initiator, sharing a

common port. This choice is further reinforced by the lack of documentation regarding those devices. Revisions will be required under a different use case, or if these peripherals indeed affect the considered applications.

The ARM Cortex-A72 Cluster on the BCM2711 does feature 4 cores, each with its own Level 1 (L1) caches for instruction (ICache), data (DCache), and page descriptors (TLBs). The L2 memory subsystem is shared between the cores, featuring a unified L2 Cache, a Snoop Control Unit for memory coherency, and the L2 prefetcher. Note that while there are two levels of TLB per core, we elect to conservatively model them as a single target. The TLBs act as single entry block; the only path to the L2 is through the L1, and the only path from the L1 goes through the L2 [23].

The interference analysis, without any additional mitigation from the hardware or software configurations, highlights the shared resources as sources of interference: the L2 subsystem, the AMBA Bus, and the LPDDR. The analysis also flags private caches as a source of interference. The L1 and L2 caches are inclusive, such that a line in the L1 must also be present in the L2. An eviction from the L2 may thus result in the same line being evicted from the L1. Without any additional restrictions, a private L1 may thus be impacted by other cores, or the L2 prefetch. Similarly memory coherency, enforced by the snoop, may result in the invalidation of data in the L1 and L2 caches due to requests from other cores, or from devices through the coherence ACE and AXI ports.

4.2. Interference Modeling using Machine Learning

We build on the microbenchmarking method of Courtaud et al. [12] to quantify how shared-memory pressure affects execution time. Our approach combines timing measurements, PMU events, and a dynamic extraction of the instruction trace. The goal is to identify the regions of code that suffer the highest slowdowns when sharing L2, buses, or DRAM. The final output is a *code weights vector* usable as a cost function for MLIR optimizations and as input to cgroup allocation policies.

4.2.1. Phase 1: Collecting Interference Timing

Instrumentation and execution setup. We reuse the microbenchmarks of Courtaud et al. [12] as training and validation victims. Each benchmark is executed twice: (i) isolation and (ii) concurrently with aggressors. Aggressors are handwritten stressors targeting specific microarchitectural components: one precisely loads on a specified LLC bank continuously (inspired from [5]); another saturates both the memory bus and DRAM. Each aggressor runs on a dedicated core in an infinite loop. We monitor their activity (aggressiveness) using hardware event counters (how much LLC misses we have, how much data are passing through the bus). **Timing measurement.** Each victim contains explicit observation points inserted (by hand) before and after loop nests or simple loops. At each point, we record the execution time in isolation and under interference. We repeat every run 100 times in both conditions. **Ground truth (delta).** For each observation point, we compute the slowdown:

$$\Delta = \frac{T_{\text{interference}}}{T_{\text{isolation}}}.$$

This ratio is the ground truth used to train our learning model.

4.2.2. Phase 2: Collecting Instruction and Data Adresses About the Victim

Assembly trace extraction. This phase is independent of timing measurements. Using `gdb`, we extract the full instruction trace of the victim (we represent a program by an execution trace): instruction address, instruction type, and the corresponding data address when applicable. **Trace transformation.** The trace is converted into a structured sequence:

$$\langle \text{InstrAddr},\ \text{InstrType},\ \text{DataAddr} \rangle.$$

This heavy parsing step is performed only once per program.

The combined dataset (timing deltas + instruction-level features) forms the input to our machine-learning model. In addition to extracting the instruction trace, we also use the collected data addresses to determine which LLC bank the victim accesses most. The bank is inferred from the physical-address bits associated with the LLC slices. The dominant bank is then targeted by an aggressor configured to saturate the same cache region.

4.2.3. Phase 3: Training

We train a BERT-style Transformer model. The encoder is pretrained on assembly code to learn structural and semantic patterns, following the same pretraining philosophy as CAWET [2]. We fine-tune it with pairs $(\text{execution trace}, \Delta)$ using an RMLSE loss, which penalizes underestimation more than overestimation.

During fine-tuning, we observe poor generalization when the model relies only on the delta-regression objective (42% accuracy). To investigate this behaviour, we extract the last attention matrix of the Transformer for each inference. This matrix has size $N \times N$, where N is the length of the trace. Most of the attention mass concentrates on instruction types that are not LOAD or STORE, nor on addresses. To align the model with actual memory behaviour, we replace timing by the number of L2 accesses. For each victim, we build an `L2_Access_MAP`, a vector counting the number of L2 accesses (using performance event) per region (these regions follow the observation-point boundaries). We also reduce the attention matrix into a vector by summing all elements of each row, then aggregating these values into the same regions, yielding `Attention_MAP`.

Both vectors are normalized so that every element lies in $[0, 1]$: `L2_Access_MAP` is scaled by the total number of L2 accesses, and `Attention_MAP` by its maximum value. Their dot product produces a correlation score bounded by the number of observation points N_{obs}. This score is added to the loss as an extra term, encouraging the model to align its attention with actual L2-access behaviour while predicting deltas.

Let

$$C = \text{dot}(\text{Attention\_MAP}, \text{L2\_Access\_MAP}).$$

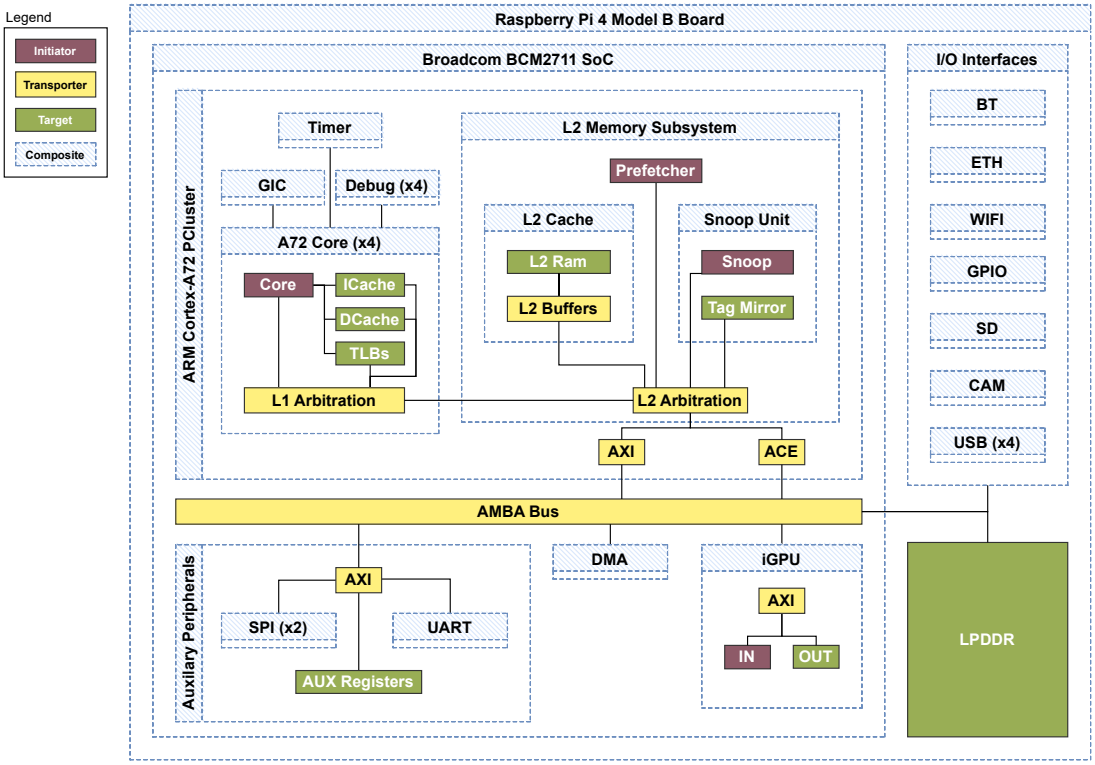

Figure 3. Overview of the PML model for the Raspberry Pi 4 Board

A perfect alignment yields $C = N_{\text{obs}}$. We convert this into a penalty by taking the distance from the ideal value:

$$\text{Penalty} = N_{\text{obs}} - C.$$

The final loss is:

$$\mathcal{L} = \text{RMLSE}(\Delta_{\text{pred}}, \Delta_{\text{obs}}) + \lambda \cdot (N_{\text{obs}} - C),$$

With RMLSE formula:

$$\text{RMLSE}(y, \hat{y}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\log(1+\hat{y}_i) - \log(1+y_i))^2}$$

where λ controls the influence of this correlation-based term. In this paper we fix λ to 0.5. With this new loss we get a higher accuracy of 72% (compared to 42% we using just RMLSE).

4.2.4. Results

Attention Matrix: With and Without the Correlation-Based Loss. Figure 4 shows a subpart of the transformer's final-layer attention matrix when trained (i) without the correlation-based penalty and (ii) with the proposed loss term. Without the penalty, the model concentrates most attention on instruction types, exhibiting little sensitivity to memory behaviour. With the penalty, attention shifts toward instruction regions that correspond to heavy L2 activity, indicating that the model better captures the actual sources of interference.

Accuracy Under Varying K. To assess how the model generalizes, we experiments on matrix multiplication, we vary the parameter K in the matrix multiplication kernel while keeping N and M fixed. Increasing K changes the stride and reuse distance of the matrix B, directly affecting the L2 pressure and interference sensitivity.

Figure 5 reports the accuracy of the prediction for different values of K. We measure the ratio (Delta) when running under maximum interferences and when running the matrix multiplication without running our aggressors. We also give the average of the ratio estimated by our Transformer model. The results show that for matrix multiplication under variation of K, our estimations never underestimate the delta while keeping a reasonable distance from the measurements. This will help us to use the machine learning model during compilation as a cost function to indicate the sensitivity of the program to interferences.

At this stage, our approach operates exclusively on binary code, which limits its applicability to MLIR-based workflows. A key direction for future work is to extend the front-end to MLIR and its relevant dialects, and to train the Transformer directly on this intermediate representation so that interference patterns can be learned and mapped at the MLIR level rather than from low-level assembly.

4.3. Interference reduction via compilation

An application running on a multi-core system may suffer from interference, primarily due to memory access contention with shared hardware resources. A substantial amount of the memory interference potential within an application is determined at compile time. Compile-time transformations to reduce cache misses and exploit data locality are good ways to reduce memory interference within a single-threaded application. However, they fall short in the context of applications that exploit parallelism and have an inherent dataflow graph.

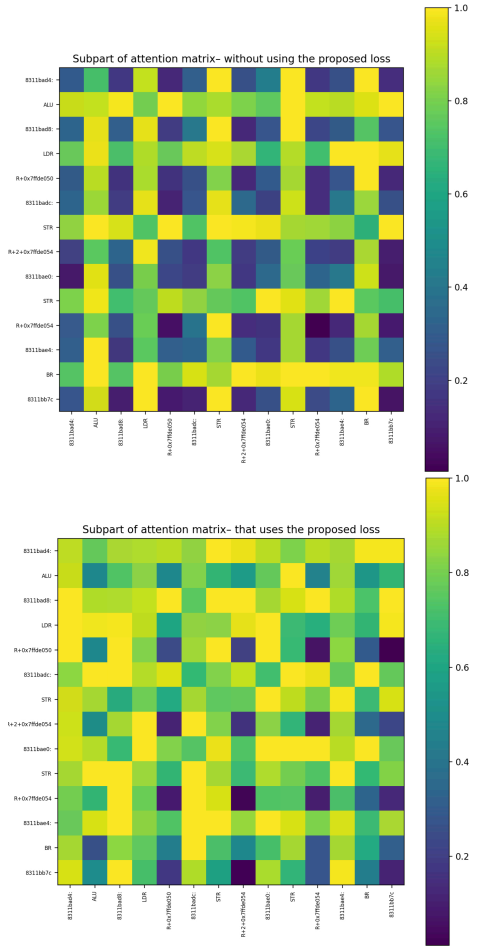

Figure 4. Final-layer attention matrices without (top) and with (bottom) the correlation loss.

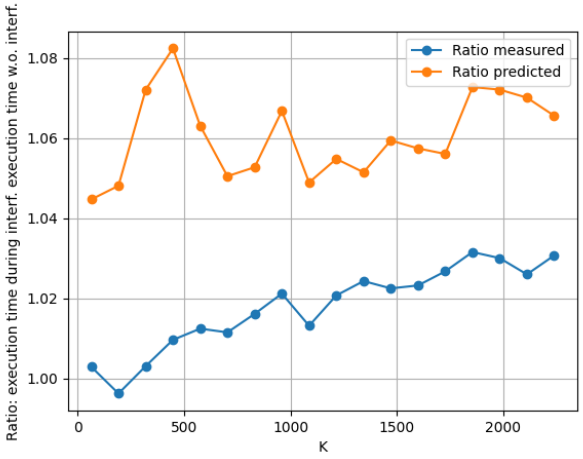

Figure 5. Prediction accuracy when varying parameter K in the matrix multiplication kernel.

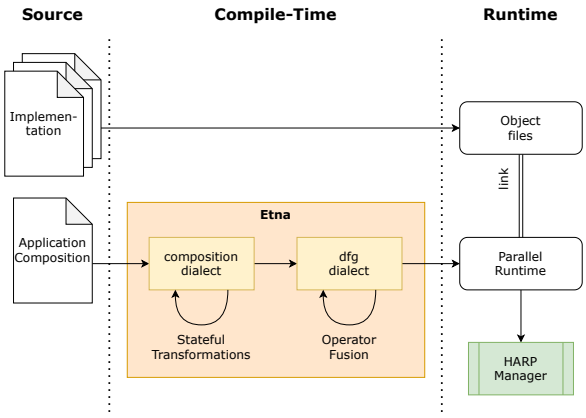

Figure 6. Overview of the compilation flow.

As part of our methodology, we aim to close this gap by providing means to reduce the interference within a parallel application on a dataflow graph level, using a combined compile time and run time approach.

4.3.1. Interference Model

In parallel dataflow applications, task switching acts as an additional source of memory interference not caught by classic single-thread focused optimizations.

Dividing an application into several threads executing in parallel often leads to an over-saturation of the underlying hardware threads with work. The result is task switching, which incurs a significant memory interference overhead, as the data gets loaded in and out of cache every time a switch occurs. However, since in a normal execution environment, the operating system scheduler allocates resources, task switching itself is unpredictable and adapting your application to the constantly changing available resources would require significant changes to user code.

Applications modeling a dataflow graph tend to be especially affected by the over-saturation problem. Even simple computational graphs often have more nodes than the target hardware has threads. However, these graphs offer the advantage of consisting of compartmentalized, separate tasks that communicate via a well-defined interface. Yet, not all nodes in a graph have the same complexity and worst-case execution time. Thus, fusing neighboring nodes of the graph can be a means to reduce memory interference by reducing the over-saturation of threads and eliminate overhead.

To address this problem, we propose a set of compile-time optimizations (subsubsection 4.3.2) to alleviate the over-saturation problem. A run-time resource manager (subsubsection 4.3.3) complementary to our cgroups method (subsection 4.4) allows the application to intelligently adjust at runtime to changes in the available resources. Our system is shown in Figure 6.

4.3.2. Compile-Time Optimizations

The core of our dataflow compilation flow is built atop *Etna* [31], a compiler toolchain based on MLIR [20]. It ingests the description of an application as a sequential composition of functions and derives a dataflow graph from it [32]. For this graph, a deterministically parallel runtime is generated, exploiting both pipeline and data parallelism.

Our compiler frontend processes an application composition expressed in a high-level syntax modeled after the programming language C. It is used to compose the high-level algorithm of the application from functions implemented in other source files of the code base. Besides function definition, declaration and calls, the syntax supports C-style loops and re-

cursion.The `composition` MLIR dialect models this representation on an IR level. It is used to understand the parallelism opportunities within an application by also modeling the mutability of data. With this information, we can apply IR transformations that allow the exploitation of data parallelism throughout the application where possible.

This representation is translated into the `dfg` dialct, which models the graph using dedicated IR constructs for dataflow nodes and edges [31, 6]. The underlying idea of this model is, that every single node will be mapped to an OpenMP [13] task. As a result of that, the fusion of nodes at a `dfg` level will immediately lead to a reduction of task pressure. Since the dialect offers a clean definition of the nodes and their communication patterns, two nodes can be fused in a safe and side-effect free fashion. Two nodes are eligible for fusion if they are executing at the same rate[1] and connected via a dataflow edge that is triggered unconditionally. We use the model presented in subsection 4.2 to identify nodes suitable for fusion.

At the end of the compilation process, we lower our `dfg` representation into the `llvm` dialect and link the generated object file against the other object files produced from the remaining code of the developer. Using our compilation flow, we generate a deterministic parallel runtime, optimized for providing better performance[32], while also reducing memory interference.

4.3.3. Run-Time Management

In order to mitigate the memory interference caused by task switching at runtime, as well as the constantly changing availability of hardware resources, a method to influence the resource assignment within the application is necessary. Depending on the available resources, nodes with higher demand (driven by computation time and cache usage) should be prioritized.

For this, we leverage HARP [30], a Linux-integrated resource management approach. It provides a unified resource allocation interface and jointly manages all application threads for an efficient resource utilization. Its support for the OpenMP programming model means that we can leverage it without needing to change any user code. Originally, this work was geared towards heterogeneous processors and improving energy efficiency. It analyzes on-line performance metrics of an application to adjust resource allocation decisions at runtime. For this work, we introduce the metric of cache misses into the decision making process of resource assignment. The more cache misses a thread shows due to task switching, the higher its priority in the resource assignment to avoid costly interference.

This runtime system is complementary to our cgroups approach, as it works exclusively on an application-thread level, rather than across processes. It can adapt the application as a whole to the number of available threads and prioritize individual nodes with high interference potential.

[1] In KPN networks, this would be labelled as a 1:1 connection.

4.4. Interference reduction via Cgroups

Linux control groups (cgroups) are a kernel feature that enables hierarchical partitioning of system resources, such as CPU, memory, and I/O, among groups of processes or tasks. They provide isolation and limit enforcement, making them suitable for mitigating interferences in multi-core environments by controlling resource allocation and task execution behaviour [16]. In our methodology, cgroups serve as a runtime mechanism to complement compile-time optimizations (e.g., via MLIR), enforcing hardware-level constraints to reduce memory contention in safety-critical systems. The context for this cgroup-based mitigation is the broader interference-aware design flow (Figure 1), where micro-architectural interferences identified by PHYLOG and machine learning (Sections 2.1 and 2.2) inform code transformations and resource configurations. Unlike prior applications of cgroups that focus on temporal isolation [10, 3] or frequency scaling [17], our approach targets memory interferences by statically configuring cgroups to prevent concurrent execution of contending tasks, enhancing predictability for certification under standards like DO-178C.

4.4.1. System Model and Assumptions

We model the system as a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ deployed on a multi-core platform, where each τ_i is provided as a binary executable. Interferences are captured in a symmetric matrix $I \in \mathbb{R}^{n \times n}$, where entry I_{ij} quantifies the degradation factor (e.g., slowdown in execution time or increase in cache-miss rate) when τ_i and τ_j run concurrently. A value $I_{ij} > 1$ indicates contention, with higher values signifying greater sensitivity (e.g., $I_{ij} = 2.5$ implies a $2.5\times$ slowdown).

Assumptions include:

- Availability of the interference matrix I, derived from upstream analyses (e.g., PHYLOG micro-benchmarks and machine learning heatmaps). In this work, we treat I_{ij} as a scalar representing the worst-case degradation across the entire execution. This assumption can be refined in future work by replacing each I_{ij} with a vector $I_{ij} = [I_{ij}^{(1)}, \ldots, I_{ij}^{(k)}]$, where each component captures the interference between specific code regions of τ_i and τ_j, enabling finer-grained mitigation strategies
- Support for cgroup v2 controllers: `cpu` (for scheduling weights and shares), `cpuset` (for CPU affinity and core pinning) and `freezer` (for pausing task execution)
- Static deployment: Configurations are determined pre-runtime

The goal is to generate a cgroup hierarchy that minimizes aggregate degradation by isolating high-interference pairs, thereby controlling task concurrency.

4.4.2. Heuristic for Cgroup Deployment

We employ a static heuristic to transform I into a cgroup deployment plan. The algorithm identifies contention-sensitive pairs or clusters (where $I_{ij} > \theta$, with θ a user-defined threshold, e.g., 1.5) and assigns them to separate sub-cgroups under a root cgroup. This enables serialization via the `freezer` controller, while global limits at the root enforce overall constraints.

The pseudocode of the heuristic is as follows:

Algorithm 1 Static Cgroup Hierarchy Construction

Require: Taskset $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, interference matrix I, threshold $\theta > 1$
Ensure: No pair with $I_{ij} > \theta$ can run concurrently
1: All tasks initially reside in the root cgroup (default kernel behaviour)
2: **for** each unique task pair $\{\tau_i, \tau_j\}$ with $i < j$ and $I_{ij} > \theta$ **do**
3: Create sub-cgroup C_i (if not already existing) and move τ_i into C_i
4: Create sub-cgroup $C_j \neq C_i$ (if not already existing) and move τ_j into C_j
5: Enable the freezer controller on both C_i and C_j
6: **end for**
7: Tasks without any high-contention pair may remain in the root cgroup

4.4.3. Experimental Evaluation

To validate the effectiveness of our cgroup-based interference mitigation strategy, we conduct a series of experiments, the goal is to demonstrate that our static cgroup hierarchy and runtime freezer mechanism can effectively restore temporal predictability in the presence of heavy memory contention.

Experimental Setup: The experiments are performed on a Raspberry Pi 4 (Broadcom BCM2711, Quad-core Cortex-A72). The system runs Raspberry Pi OS Lite with Cgroup v2 enabled. We utilize the RT-Bench framework [24], which extends the TACleBench suite for periodic real-time execution. Two tasks constitute our workload:

- We use `matrix1` as a **victim task** (τ_{victim}), a $256 \times 4096 \times 9$ integer matrix multiplication program. Its ≈4MB working set exceeds L2 capacity, making it sensitive to cache evictions. It has a period of $T = 100$ ms and a relative deadline of $D = 30$ ms to simulate a tight real-time constraint.
- **Contender Task (τ_{noise}):** We use the `bandwidth` benchmark from the IsolBench suite as the interference source. This synthetic benchmark issues sequential writes to a large buffer, saturating memory bandwidth. It has a period of $T = 200$ ms and a relative deadline of $D = 200$ ms.

To maximize resource contention, both the victim task and the interfering workload are pinned to different physical cores (which are also isolated from the OS execution) using the `cpuset` controller. This configuration forces the tasks to compete for CPU, L1/L2 caches, and memory bandwidth.

We evaluate our workload in three execution scenarios over 100 periodic jobs:

1. **Solo:** τ_{victim} runs in isolation to establish a baseline worst-case execution time (WCET).
2. **Interference (Unprotected):** τ_{victim} and τ_{noise} run concurrently in the same core without any mitigation.
3. **Protected:** τ_{victim} and τ_{noise} are placed in separate cgroups inside the same core as per Algorithm 1. A userspace monitor program executed in another core polls the CPU usage of the cgroups (at 50ms intervals) and freezes the τ_{noise} group when the CPU bandwidth consumption exceeds a safety threshold (30%), effectively serializing access during peak contention.

Results and Analysis: The results of our evaluation are summarized in Figures 7, 8, and 9.

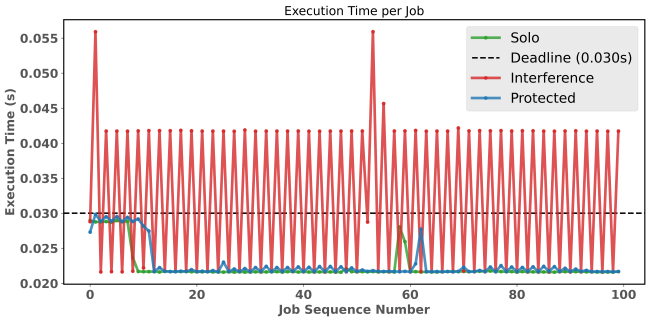

Figure 7. Job-wise execution times for `matrix1`. The dashed line represents the 30ms deadline

Figure 7 plots the execution time for each of the 100 jobs. In the *Solo* case (green), execution time is stable and well below the 30ms deadline. In the *Interference* scenario (red), memory contention causes severe execution time spikes overlapping every two consecutive jobs, frequently exceeding the deadline by up to 2×. In the *Protected* scenario (blue), the freezer mechanism successfully detects contention and pauses the aggressor.

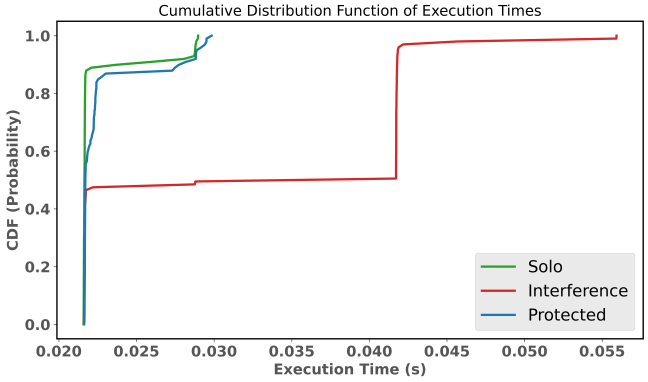

Figure 8. Cumulative Distribution Function (CDF) of execution times.

The Cumulative Distribution Function (CDF) in Figure 8 further illustrates the impact of our approach on the probability of jobs satisfying their performance deadlines. The *Interference* curve indicates that a significant portion of jobs suffer from extreme delays. The *Protected* curve is steep and closely aligns with the *Solo* baseline, demonstrating that our approach effectively eliminates the worst-case outliers caused by memory interference.

Finally, Figure 9 quantifies the number of jobs that our victim task suffers in each execution scenario. In the *Solo* exe-

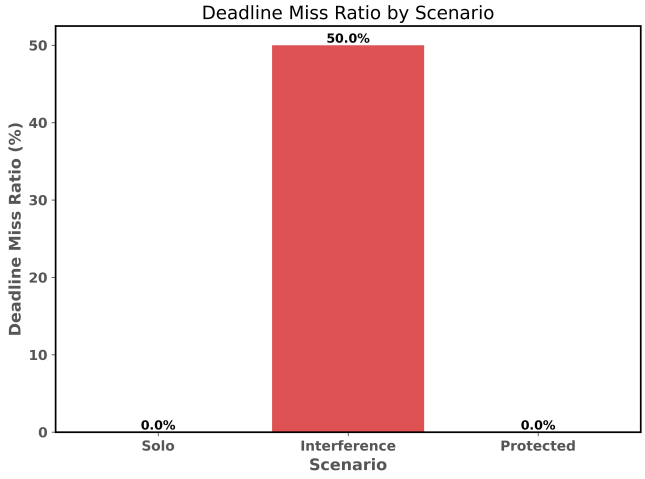

Figure 9. Deadline Miss Ratio comparison.

cution scenario, τ_{victim} meets all deadlines as expected. Under *Interference* execution, approximately 50% of jobs miss their deadline. This ratio is consistent with the period relationship: τ_{victim} ($T = 100\,\text{ms}$) and τ_{noise} ($T = 200\,\text{ms}$) overlap on every second job of τ_{victim}, causing contention induced slowdown precisely when both tasks execute concurrently. Such a miss rate is unacceptable for safety-critical applications. On the other hand, the *Protected* configuration eliminates all deadline misses by freezing τ_{noise} during periods of high contention exceeding the allowed threshold, ensuring τ_{victim} completes within its deadline..

5. Related work

The survey by Lugo et al. [21] categorizes interference mitigation techniques based on the targeted shared resource (e.g. memory, cache, bus) and their integration into the schedulability analysis. Maiza et al. [22] provide a complementary taxonomy focused on timing verification, distinguishing between isolation-based and contention-aware approaches, including hardware partitioning, scheduling strategies, and hybrid solutions. While prior approaches often target individual aspects of interference, our work introduces a unified methodology that encompasses both hardware and software dimensions to enable integrated analysis and mitigation.

Prior work has addressed the challenge of predicting multicore contention impact on program execution time, primarily by learning a global contention ratio for whole programs. Brando et al. [8] proposed the use of Quantile Regression Neural Networks to estimate the contention-induced slowdown. Their model leverages event monitors collected during isolated execution to predict this delta value for each program. By tuning the quantile parameter, their model aim to reduce underestimations and provide safer timing budgets during early design phases. Courtaud et al. [12] approached the problem from a different angle. They introduced a rich set of microbenchmarks to emulate diverse memory behaviours and revealed the limitations of purely bandwidth-based characterizations. By profiling qualitative and quantitative memory features using Valgrind, they trained a random forest model to predict memory contention overheads. While both works significantly improve the accuracy of contention ratio prediction at the whole-program level, they do not address the localization of contention effects within the code. In contrast, our work formulates the problem as an inverse machine learning task: given global contention observations and static/cache analysis for each instruction, we aim to identify the code regions most susceptible to shared resource contention. This finer granularity enables targeted optimizations and better root-cause analysis of timing variations under interference.

Prior work on dataflow graph fusion like TileFlow [36] and FuseFlow [19] has been focused on operator fusion in the context of DNN models deployed on accelerators. These frameworks are more focused on reducing the amount of memory transfers via fusion, whereas our approach is focused on optimizing the resource usage for more constrained, embedded platforms. MAESTRO [18] describes dataflow through data-centric notations, calculating performance metrics through iteration analysis rather than our model-inference approach.

Other works on on-line interference analyses for multithreaded programs [35] focus on modeling the execution of an application and predicting interference. This data is then used to influence the scheduling decisions by the hypervisor running all applications. This is in contrast to our approach that focuses on a deterministic execution model and on-line adjustments of the resources made available by the underlying system. The system described by the authors can explicitly not handle applications with an over-saturation of threads.

Techniques that focus on optimizing the cache layout of multithreaded applications [29] have also shown promising results. However, they are focused on optimizing cache usage alone, neglecting the overhead introduced by task switching and the changes this brings to the cache layout.

Linux cgroups provide a flexible framework for resource management and have been adapted to address challenges across multiple domains.

In the context of real-time systems, Chen et al. [10, 11] proposed SchedGuard, a temporal protection framework that uses cgroups to prevent untrusted tasks from executing during specific time segments, protecting against scheduler-based side-channel attacks. Their approach demonstrates the effectiveness of cgroup-based mechanisms for enforcing execution constraints. Similarly, our work exploits cgroups to control task execution, although we focus on mitigating memory contention rather than timing-based security threats.

Andriaccio et al. [3] present a cgroup-based real-time scheduler integrated with the Linux deadline server infrastructure to enforce temporal isolation in IoT and edge environments. By associating deadline servers with cgroups, they reserve runtime Q and period P for groups of fixed-priority tasks, supporting multicore migrations to maintain schedulability under multiprocessor models. Validated on FastFlow streaming applications, it achieves lower response-time variance and higher throughput than default schedulers. Unlike their focus on timing predictability for dynamic IoT workloads, our method leverages cgroups for memory-centric isolation, monitoring statistics to trigger freezes and serialize contending tasks, thereby addressing contention vulnerabilities without relying on bandwidth reservations.

For mixed-criticality systems, Kim et al. [17] used cgroups to separate critical and non-critical tasks, then employed the CPUFreq governor to throttle non-critical cgroups when memory contention was predicted, indirectly reducing memory bandwidth consumption through CPU frequency scaling. In contrast, our work directly prevents co-execution of memory-intensive task pairs using the cgroup freeze mechanism, avoiding the need for frequency scaling and its associated performance overhead on non-contending tasks.

Beyond the real-time domain, cgroups are extensively applied in cloud computing and virtualized environments, recent work by Volpert et al. [33, 34] examines noisy-neighbor effects that persist despite CPU isolation using cgroups. They present a workload-agnostic, online detection technique that instruments the Linux scheduler with eBPF to collect runtime metrics—notably scheduling latency and preemption frequency—and use those signals to identify interference between supposedly isolated cgroups. Their results show that scheduler-level interactions can break the practical isolation guarantees of cgroups, motivating adaptive, runtime-aware countermeasures. Building on these findings, our contribution extends cgroups beyond static limits by combining runtime contention monitoring and reactively prevent concurrently running task pairs when measured interference exceeds configurable thresholds.

6. Conclusion & Future Work

In this paper, we presented the initial steps of an interference-aware design methodology for high-performance multi-core systems. The current results are preliminary but they outline a workflow that combines formal analysis, profiling, and system-level control to mitigate memory interference. Future work will investigate the scalability of the approach, its robustness across diverse workloads and hardware platforms, and its applicability to heterogeneous CPU–GPU and NUMA architectures.

Acknowledgement

This work is partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through the InterMCore project (Grant No. 505744711) and by the Agence Nationale de la Recherche (ANR) under Grant No. ANR-22-CE92-0066.

References

[1] Alif Ahmed and Kevin Skadron. Hopscotch: a micro-benchmark suite for memory performance evaluation. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 167–172, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372060.

[2] Abderaouf N Amalou, Elisa Fromont, and Isabelle Puaut. Cawet: Context-aware worst-case execution time estimation using transformers. In *ECRTS 2023-35th Euromicro Conference on Real-Time Systems*, volume 262, pages 7–1. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[3] Yuri Andriaccio, Luca Abeni, and Massimo Torquati. Scheduling iot applications in real-time control groups. In *2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 01–08. IEEE, 2025.

[4] Simon Arridge, Peter Maass, Ozan Öktem, and Carola-Bibiane Schönlieb. Solving inverse problems using data-driven models. *Acta Numerica*, 28:1–174, 2019.

[5] Michael Bechtel and Heechul Yun. Cache bank-aware denial-of-service attacks on multicore arm processors. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 198–208. IEEE, 2023.

[6] Jiahong Bi, Guilherme Korol, and Jeronimo Castrillon. Leveraging the mlir infrastructure for the computing continuum, September 2024. URL https://doi.org/10.5281/zenodo.13898631.

[7] Frédéric Boniol, Youcef Bouchebaba, Julien Brunel, Kevin Delmas, Thomas Loquen, Alfonso Mascarenas Gonzalez, Claire Pagetti, Thomas Polacsek, and Nathanaël Sensfelder. PHYLOG certification methodology: a sane way to embed multi-core processors. In *10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)*, Toulouse, France, January 2020. URL https://hal.science/hal-02441323.

[8] Axel Brando, Isabel Serra, Enrico Mezzetti, Jaume Abella, and Francisco J Cazorla. Using quantile regression in neural networks for contention prediction in multicore processors. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[9] Certification Authorities Software Team (CAST). CAST-32A: Position Paper on Multi-core Processors, November 2016.

[10] Jiyang Chen, Tomasz Kloda, Ayoosh Bansal, Rohan Tabish, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. Schedguard: Protecting against schedule leaks using linux containers. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 14–26. IEEE, 2021.

[11] Jiyang Chen, Tomasz Kloda, Rohan Tabish, Ayoosh Bansal, Chien-Ying Chen, Bo Liu, Sibin Mohan, Marco Caccamo, and Lui Sha. Schedguard++: Protecting against schedule leaks using linux containers on multicore processors. *ACM Transactions on Cyber-Physical Systems*, 7(1):1–25, 2023.

[12] Cédric Courtaud, Julien Sopena, Gilles Muller, and Daniel Gracia Pérez. Improving prediction accuracy of memory interferences for multicore platforms. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 246–259. IEEE, 2019.

[13] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[14] EASA. AMC (Acceptable Means of Compliance) 20-193 on the use of multi-core processors (MCPs), 2020.

[15] International Organization for Standardization (ISO). ISO 26262: Road Vehicles – Functional Safety, 2018. Second Edition, Parts 1–12.

[16] Tejun Heo. Control group v2 — the linux kernel documentation, October 2015. Accessed: 2025-11-18.

[17] Jungho Kim, Philkyue Shin, Soonhyun Noh, Daesik Ham, and Seongsoo Hong. Reducing memory interference latency of safety-critical applications via memory request throttling and linux cgroup. In *2018 31st IEEE International System-on-Chip Conference (SOCC)*, pages 215–220. IEEE, 2018.

[18] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings. *IEEE micro*, 40(3):20–29, 2020.

[19] Rubens Lacouture, Nathan Zhang, Ritvik Sharma, Marco Siracusa, Fredrik Kjolstad, Kunle Olukotun, and Olivia Hsu. Fuseflow: A fusion-centric compilation framework for sparse deep learning on streaming dataflow. *arXiv preprint arXiv:2511.04768*, 2025.

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '21, page 2–14, Seoul, Korea (South), 2021. IEEE Press. ISBN 9781728186139.

[21] 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.

[22] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Computing Surveys (CSUR)*, 52(3):1–38, 2019.

[23] Alfonso Mascareñas-González, Frédéric Boniol, Benjamin Lesage, and Claire Pagetti. Towards a validated core memory model through (mp)soc events. In *2025 28th International Symposium on Real-Time Distributed Computing (ISORC)*, 2025. doi: 10.1109/ISORC65339.2025.00027.

[24] 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*, pages 184–195, 2022.

[25] Raspberry Pi. ARM Cortex-A72 MPCore Processor Technical Reference Manual –r0p3, December 2016. URL https://developer.arm.com/documentation/100095/0003/?lang=en.

[26] Raspberry Pi. BCM2711 ARM Peripherals, June 2022. URL https://pip-assets.raspberrypi.com/categories/545-raspberry-pi-4-model-b/documents/RP-008248-DS-1-bcm2711-peripherals.pdf.

[27] Raspberry Pi. Raspberry Pi 4 Model B – Datasheet. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf, March 2024.

[28] RTCA, Inc. and EUROCAE. DO-178C / ED-12C: Software Considerations in Airborne Systems and Equipment Certification. RTCA, Inc. and EUROCAE, 2011. RTCA document DO-178C, EUROCAE document ED-12C.

[29] Subhradyuti Sarkar and Dean M Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 353–368. Springer, 2008.

[30] Till Smejkal, Robert Khasanov, Jeronimo Castrillon, and Hermann Härtig. HARP: Energy-aware and adaptive management of heterogeneous processors. In *Proceedings 26th ACM/IFIP International Middleware Conference (Middleware'25)*, Middleware '25, New York, NY, USA, December 2025. Association for Computing Machinery.

[31] Stephanie Soldavini, Felix Suchert, Serena Curzel, Michele Fiorito, Karl Friebel, Fabrizio Ferrandi, Radim Cmar, Jeronimo Castrillon, and Christian Pilato. Etna: Mlir-based system-level design and optimization for transparent application execution on cpu-fpga nodes. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 224–224. IEEE, 2024.

[32] Felix Suchert, Lisza Zeidler, Jeronimo Castrillon, and Sebastian Ertel. Condrust: Scalable deterministic concurrency from verifiable rust programs. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, pages 33–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.

[33] Simon Volpert, Sascha Winkelhofer, Daniel Seybold, Jörg Domaschka, and Stefan Wesner. The hidden costs of shared cpu resources: A closer look at cgroups and qos. In *Softwaretechnik-Trends Band 44, Heft 4*. Gesellschaft für Informatik eV, 2024.

[34] Simon Volpert, Sascha Winkelhofer, Jörg Domaschka, and Stefan Wesner. Detecting noisy neighbors in cpu-isolated cgroups environments. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, pages 224–231, 2025.

[35] Yong Zhao, Jia Rao, and Qing Yi. Characterizing and optimizing the performance of multithreaded programs under interference. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 287–297, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341219. doi: 10.1145/2967938.2967939. URL https://doi.org/10.1145/2967938.2967939.

[36] Size Zheng, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, and Yun Liang. Tileflow: A framework for modeling fusion dataflow via tree-based analysis. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1271–1288, 2023.