Practical Recommendations for Accurately Predicting Performance Degradation Caused by Memory Contention

Bruno Dzikowski
bruno.dzikowski@student.uva.nl

November 28, 2025, 66 pages

Academic supervisor: Benny Åkesson, k.b.akesson@uva.nl
Daily supervisor: Ben Pronk, ben.pronk@tno.nl
Host organisation: TNO-ESI, https://esi.nl

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

As companies developing cyber-physical systems (CPS) increasingly adopt microservice-based architectures deployed on commercial-off-the-shelf multi-core platforms, there arises a need for robust performance prediction models to validate candidate deployments. Multiple microservices are often scheduled on the same physical machine to improve utilization, but this can induce competition for shared hardware resources. This phenomenon, known as *resource contention*, has a major influence on application performance, and therefore must be addressed in order to maintain prediction accuracy. Existing methods model resource contention by independently characterizing application *sensitivity* (i.e. susceptibility to competition) and *contentiousness* (i.e. intensity of pressure generated in shared resources), and then combining those characteristics to estimate performance. These approaches scale well with the combinatorial explosion of possible deployments. However, they do not provide concrete guidelines for implementing the contentiousness characterization mechanism or for configuring the OS and hardware to achieve accurate prediction. Collectively, these shortcomings hinder their practical adoption in the CPS domain.

In this thesis, we propose a performance prediction framework that extends an existing methodology by introducing three main contributions: (1) empirically validated design of a contentiousness profiling component, (2) a quantitative analysis of how system configuration affects prediction accuracy, and (3) design and implementation of an experimental testbed. We validate our approach in a series of controlled experiments across 195 distinct co-location scenarios involving 22 applications, including three latency-critical microservices. Our findings confirm that our contributions help reduce prediction error. Overall, the proposed method achieves strong accuracy (median absolute error $\approx$ 1.4%, 90th percentile absolute error $\approx$ 7.4%), demonstrating its efficacy for performance forecasting of microservice deployments on multi-core platforms.

Contents

Chapter 1

Introduction

The complexity of high-tech equipment is rising [1]. Cyber-Physical Systems (CPS) like photolithography machines and self-driving cars can contain millions of lines of code [2, 3]. Moreover, many customers in the industry increasingly expect software systems to be tailored to their specific needs [1]. As a result, companies must offer a broad array of unique system configurations. Effectively developing and maintaining these diverse and complex systems presents a significant challenge — one that demands a paradigm shift in CPS development.

To address this issue, companies are increasingly adopting a *platform-based* approach, i.e. they start creating reusable software components that can be composed into highly customized systems, with a goal of reducing both the time to market and cost of new software. Microservices are particularly well-suited to implement this development model, because they are loosely coupled, use lightweight communication protocols, and are independently deployable. This causes the microservice architecture to rapidly gain traction in the CPS industry.

Due to increasing shortage of skilled engineers [4], frameworks capable of automating the integration and deployment of microservice-backed systems based on functional and non-functional requirements became highly desirable in the industry. To satisfy the non-functional requirements, these frameworks must support optimization of multi-dimensional objectives such as hardware cost and runtime performance. This process is difficult due to the combinatorial explosion of possible deployments: the asymptotic time complexity of deploying the microservices in every possible combination is $O(N^M)$, where N denotes the number of deployment nodes and M the number of microservices. As a result, deploying the microservices and measuring their performance is prohibitively time-consuming for large systems, motivating the need for techniques that can accurately estimate their performance without instantiating every runtime combination of applications.

1.1 Problem Statement

Scalable performance prediction for microservice-based CPSs can be achieved using a compositional analytical approach, such as that proposed by Volaard [5]. These approaches reduce the number of required deployments from $O(N^M)$ to $O(M)$ by constructing independent performance models for each microservice, and then combining them using a computationally inexpensive operation to predict the performance of all possible configurations. The method of Volaard achieves this by monitoring the performance of individual microservices over an extended period of time and then aggregating request-processing and network-call durations to produce an overall performance estimate for the entire system. However, this approach does not model *resource contention*, which occurs when applications that are deployed on the same physical machine compete for shared computing resources like CPU, memory, network and disk. Such contention can cause the performance of applications to degrade by as much

as 35% [6], and render any contention-unaware method significantly inaccurate.

Several existing compositional prediction methods also offer scalable prediction of performance, while addressing the problem of contention. A large set of those methodologies (e.g. [6–12]), which are particularly promising for CPS deployment validation, predict performance degradation by independently characterizing each application in terms of *sensitivity* (i.e. susceptibility to competition) and *contentiousness* (i.e. intensity of pressure generated in shared resources), and then combining those characteristics to estimate performance. However, many are tailored to domains other than microservices or exhibit key limitations, such as: supporting no more than two co-located applications, requiring lengthy profiling, or depending on large volumes of live operational data. Furthermore they lack guidelines on how to design of a mechanism for characterizing contentiousness of applications, and recommendations on how to configure an experimental environment that allows for accurate prediction of performance degradation. Our findings suggest that these factors can substantially affect accuracy, increasing the median prediction error by as much as 11%. To address these shortcomings, we propose extending an existing method to develop a complete foundational framework for resource contention prediction in microservice-based CPSs, supported by experimental evaluation and an open-source implementation.

1.2 Scope

We limit the scope of our thesis with the following assumptions:

- We only model contention in the memory subsystem. In our experiments, we purposefully isolate the used workloads from contention in the CPU cores, and modify the boundaries of measured operations to exclude network latency.
- We assume that all compute nodes are homogeneous, and we use identical machines in our experiments. We expect our approach to generalize to heterogeneous execution platforms by repeating characterization on every architecture in the environment. However, we do not perform any additional experiments to confirm this assumption and investigate the possible impact on prediction accuracy.
- We make an assumption that all microservices and workloads receive identical inputs for each execution and that their performance is approximately stable across profiled operations.
- Our prediction of performance degradation is restricted to two co-running applications. We leave generalizing the method to larger co-locations for future research.

We further discuss addressing these limitations in future work in Section 9.3.

1.3 Research Questions

We determine the work of Mars et al. [6] as the best research method to extend to meet our requirements after a literature review in Chapter 3. Informed by the core principles of this approach and our problem statement, we pose the following research questions for this thesis:

> **RQ1:** *"How to design a mechanism for accurately characterizing contentiousness of applications?"*
> **RQ2:** *"How to configure an execution environment that allows for accurate prediction of performance degradation caused by memory subsystem contention?"*

1.4 Contributions

In the process of researching the open questions mentioned in Section 1.3, this thesis makes the following novel contributions:

(C1) Design of contentiousness scoring mechanism We propose four mechanisms that exploit different memory access patterns. Using a series of controlled experiments, we empirically evaluate their effects on prediction accuracy and share our recommendations on which mechanism performs best.

(C2) Recommendations for configuring the execution environment We conduct a literature review to identify OS features and workload isolation practices possibly affecting performance measurements. We then empirically evaluate the effects of two of identified features, CPU frequency scaling governors and Turbo Boost, on the accuracy of our methodology. Finally, we distill actionable recommendations for practitioners based on our findings.

(C3) Design and implementation of a contention research framework We propose example implementations of synthetic contention generators and contentiousness scoring workloads. Additionally, we design an experimental testbed capable of predicting performance degradation of both native Linux applications and Kubernetes-deployed microservices, and validating the results of prediction. We also publicly release our implementation under the MIT open source license[1].

1.5 Thesis Outline

The remainder of this thesis is organized as follows. First, we provide necessary background information in Chapter 2. Next, in Chapter 3, we perform a literature review in in order to find a contention prediction method that is suitable as a base for our purposes, and discuss it in Chapter 4 as additional background information.

In the next three chapters, we describe our novel contributions. We establish experimentally tested design guidelines for a contentiousness scoring mechanism (C1) in Chapter 5. Additionally, we for share our recommendations for mitigating the influence of system configuration on prediction accuracy (C2) in Chapter 6. We also implement a proof-of-concept prediction framework and experimental testbed, informed by and validated through two case studies; the implementation and testbed (C3) are presented in Chapter 7.

Finally, we verify the approach in a series of controlled experiments using two corpora of applications. One corpus is an industry benchmark that is widely used in the prior research in the domain, while the second one is a subset of a microservice-backed system developed by TNO that exhibits characteristics of a real-world CPS. This evaluation is discussed in detail in Chapter 8. We conclude the thesis by answering the research questions and discussing future work in Chapter 9.

[1] `https://github.com/Bruno-Dzk/resource-contention`

Chapter 2

Background

After outlining the context and research questions in Chapter 1, we present the background material necessary to understand this work. We first describe the architecture and operating principles of a typical memory subsystem in Section 2.1. Then, we give a brief overview of performance measurement techniques in Section 2.2. Finally, we discuss the phenomenon of resource contention in multi-core systems in Section 2.3.

2.1 Memory Subsystem

As stated in Chapter 1, this thesis focuses on predicting performance degradation resulting from memory contention. Therefore, familiarity with a typical memory subsystem's structure and operating logic helps in grasping our work.

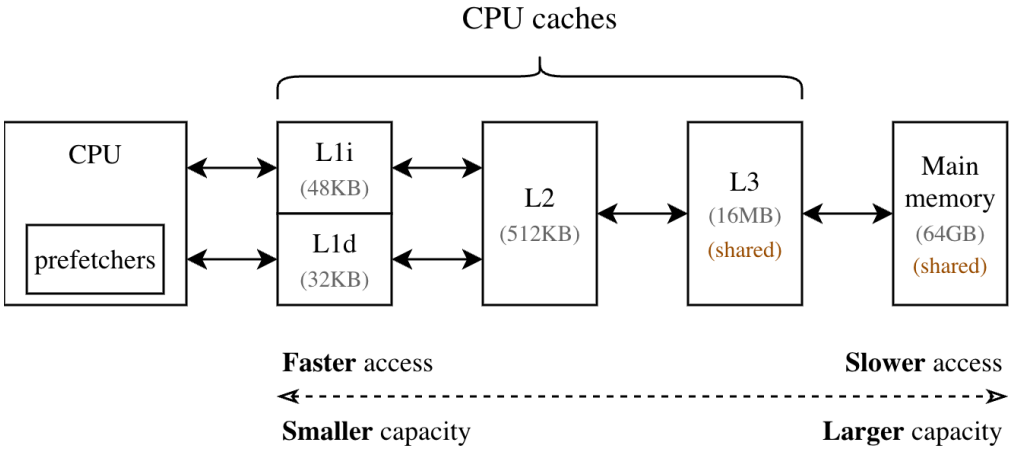


Figure 2.1: Diagram of an example memory subsystem hierarchy

As illustrated in Figure 2.1, the memory subsystem of a modern multi-core machine is typically stratified into several layers, including the CPU caches and the main DRAM memory. In this hierarchy, each successive layer offers greater storage capacity, but at the price of slower access. The reference capacities of each layer come from the machines in our experimental setup.

The first three layers of the memory hierarchy are the CPU caches. The L1 cache is the smallest and offers the fastest reads. It is usually divided into two parts: the instruction cache (L1i) and the data cache (L1d). The L2 cache is larger and slightly slower. The L3 cache has the greatest capacity but the highest latency among all cache levels, and is typically shared by all cores on a single chip.

The outermost layer of the memory subsystem is the main DRAM memory, connected to the rest of the subsystem through a memory bus. This connection enables applications to retrieve data from memory; however, it provides only limited bandwidth, which can become a performance bottleneck for memory-intensive workloads [13]. The main memory serves as a shared resource among all cores on a single processor, providing a significantly larger capacity than the caches but being much slower. In multi-socket or non-uniform memory access (NUMA) architectures, portions of the main memory are physically attached to different processors, yet remain accessible to all of them. The access time and bandwidth in such systems depend on whether the memory being accessed is local (attached to the requesting CPU) or remote (attached to another CPU), which introduces additional complexity in managing memory performance and data placement. In this thesis, we will not consider prediction on NUMA architectures, and focus solely on uniform memory access.

The arrows represent bi-directional data flow between the memory subsystem layers, which are organized from left to right based on speed and capacity. When a CPU core issues a memory access, it first probes the L1 cache. If the requested data or instruction is absent, a cache miss occurs and the request is forwarded to the L2 cache; if it is still missing, the lookup proceeds to the shared L3 cache, and so forth, until it reaches the main memory. When data or an instruction is fetched from the next (slower) layer after a cache miss, it is typically promoted to and retained in the faster cache levels to satisfy subsequent accesses; it remains there until capacity pressure and the cache's replacement policy (e.g., LRU, FIFO) cause it to be evicted. This data flow mechanism reduces latency by obtaining data from the fastest available storage level.

A *prefetcher* is an optimization mechanism that reduces program execution time by proactively loading instructions or data from slower hierarchy layers into faster ones before the CPU actually requests them. Prefetchers can be implemented in hardware, in software, or as hybrid approaches. They often leverage locality patterns in memory access behavior: *temporal locality*, characterizing repeated accesses to the same addresses over time; and *spatial locality*, exhibited by accesses to addresses that are either contiguous or separated by a constant offset. Prefetching reduces the cost of servicing memory access requests by lowering miss rates and the effective latency of memory accesses [14], improving performance of memory-intensive workloads.

2.2 Performance

Modern cyber-physical systems are often held to high performance requirements. The microservices forming such systems are commonly required to provide good enough *latency* (the elapsed time to handle an incoming request) and *throughput* (the rate of requests serviced by the service). These aspects are typically quantified by measuring the sustained request-processing rate over time to estimate throughput; and by analyzing the distribution of request handling durations, typically reporting percentiles, to summarize latency. These measures are also known as *service-level indicators*[1].

In this thesis we focus strictly on application latency, so all future references to *performance* will refer to application latency aggregated over a time window, e.g. the median latency measured over the last n minutes. During our experiments, we collect microservice latency metrics using telemetry tools such as the Prometheus monitoring system[2]. Additionally, we employ CPU-intensive benchmarks, which are executable programs that automatically report their execution duration aggregated over several runs.

[1] `https://sre.google/sre-book/service-level-objectives/`

[2] `https://prometheus.io/`

2.3 Resource Contention

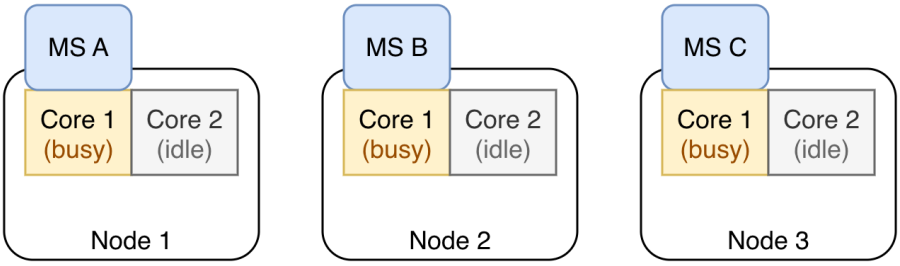
Our work revolves around the accurate prediction of the performance degradation caused by resource contention. Thus, the following subsections focus on the microservice placement strategies that induce contention, the variety of contested resources, and the impact of interference on performance.

2.3.1 Microservice Placement

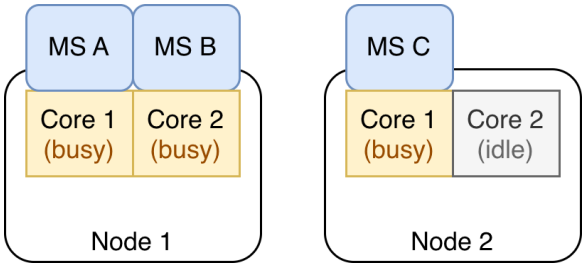
Each microservice within a software system typically consists of an application binary and configuration files that define how the binary should be deployed and executed within its target environment. When the compute infrastructure consists of multiple multi-core machines, each microservice must be assigned to a compute node as part of the *microservice placement* process. It may also be pinned to always execute a on a set of specific CPU cores, or dynamically assigned a set cores by some sort of a CPU scheduler.

Broadly speaking, there are two possible strategies when placing microservices on compute nodes. The first is to deploy them in isolation, i.e. assign a dedicated node to every microservice. This strategy offers predictable performance, since only one application runs on the machine. The second option is to *co-locate* multiple microservices, i.e. assign them to the same physical machine. This can decrease energy and hardware costs, but it can also result in resource contention.

We illustrate the microservice placement strategies in Figure 2.2. Deploying in isolation is showcased in Figure 2.2a. In this case three nodes are required, since there are three microservices. Also three cores remain idle, so the per-core utilization is 50%. In Figure 2.2b, the microservices are allowed to be deployed in co-location. One of the servers is redundant in this case, which decreases potential hardware costs. Also, only one core remains idle, which means that the utilization is at 75%, so less energy is wasted.



(a) Deploying microservices in isolation. Three servers, three idle cores, 50% utilization.



(b) Co-locating microservices. Two servers, one idle core, 75% utilization.

Figure 2.2: Two example deployments of a system containing three microservices (MS)

2.3.2 Performance Degradation

As we discussed, co-scheduling multiple applications on a single machine can decrease energy and hardware costs by improving hardware utilization. However, co-located workloads may compete for shared resources of the multi-core host system, resulting in resource contention, also known as *interference*, that deteriorates their performance [6, 10, 15]. Resource contention is reciprocal: when multiple applications execute concurrently, each may suffer slowdown due to interference from co-running neighbors, while at the same time generating resource pressure on those neighbors. The resulting performance degradation for a pair of co-running applications can be expressed formally as:

$$Deg_A(C) = \frac{P_A(C) - P_A(\varnothing)}{P_A(\varnothing)} \tag{2.1}$$

where $P_A(\varnothing)$ denotes the performance (latency) of an application under analysis A when running alone, $P_A(C)$ represents the latency of A when co-located with a competing application C, and Deg_A denotes the normalized performance loss experienced by A due to interference present in the co-location. This value is expected to lie within the closed interval $[0, 1]$. However, in cases where performance under co-location exceeds performance in isolation (e.g. due to measurement variability) it may assume a negative value.

Contested resources can include CPU, caches, memory, disk storage, and network [7, 16, 17]. As a result, even applications running on a dedicated CPU core [12], with statically partitioned memory and disk space, can still be affected by interference and, in turn, disrupt other services [8]. The following causes of resource contention have been identified in literature:

CPU interference Simultaneous multithreading techniques, such as Hyper-Threading, enable multiple threads to execute on a single physical core. Using these techniques can cause co-located applications to contend for CPU execution slots and incur slowdown [18]. In case of applications running on virtual machines (VMs), each VM is assigned a number of virtual CPU cores that the hypervisor schedules onto physical cores. If the time slices allocated to a VM for execution are too short, the performance of the applications running on the VM degrades. [17, 19]

Memory interference Multiple co-located applications can share layers of the memory subsystem, depending on the precise set of cores they are executing on [18], and the size of their working data set [6]. This sharing can cause increased rate of cache misses in shared CPU caches [6–8, 17], since concurrent workloads may collectively demand more cache capacity than is available, and cause mutual eviction of each other's cache lines. Moreover, in case of competitors that use a large portion of memory bandwidth, it can slow down memory accesses as well. Some studies suggest that memory bandwidth is the most dominant bottleneck for application performance [20].

I/O contention Applications can compete for local (e.g. disk) and remote (e.g. network) I/O resources, reducing available bandwidth and thus slowing data exchange and task execution of their neighbors [17]. In extreme cases workloads may be starved and stop functioning.

The severity of performance degradation depends on the resources the application needs to perform its functionality, as well as the intensity of pressure exerted on those resources. For example, some applications may be susceptible to memory contention, while remaining almost insensitive to network interference. This can be leveraged by co-locating workloads with disparate resource needs to minimize performance degradation [7].

Chapter 3

Related Work

In the previous chapter, we explained how microservices can interfere with each other by competing for shared resources. In this chapter, we conduct a literature survey in order to identify existing approaches for predicting performance degradation caused by resource contention. Our goal is to classify existing research in the field by key concepts, assess its relevance to the problem we aim to solve, and select one of the existing methods as a foundation for our contributions. In Section 3.1 we outline our criteria and process for selecting related work and provide a brief characterization of the existing literature. Next, we categorize reviewed works by similarities in utilized approach in Section 3.2. Finally, in Section 3.3 we outline the requirements for our problem, assess to what extent they are satisfied by existing work, and describe how our contributions close the remaining gaps by fulfilling previously unmet requirements.

3.1 Selection of Reviewed Works

Using the backwards and forwards citation search technique (a.k.a snowballing), we surveyed multiple works and selected 14 studies proposing various methods for interference prediction; Table 3.1 summarizes their key characteristics. Some approaches are narrowly specialized and can predict contention only for particular shared resources, whereas others are resource-agnostic or have been generalized to cover the full set of resources discussed in Section 2.3.2. In most cases, the primary objective is consistent: predicting the performance degradation induced by interference. Notable exceptions are the method of Delimitrou and Kozyrakis [7], which focuses on reducing contention in datacenter environments without providing precise estimates of performance, and the work of Yang et al. [9], which aims to enforce required application performance by dynamically de-scheduling competing workloads. The reviewed literature spans diverse domains, and only three studies are specifically focused on microservices.

3.2 Categorization by Approach

We identified several classes of methodologies, distinguished from one another by core principles that recur across multiple studies, and utilized these classes to categorize the reviewed literature in Table 3.2. The rest of this section presents a detailed discussion of these approaches.

3.2.1 Sources of Interference

Examples of previous work relying on tunable *sources of interference* (SoIs) include [6–9]. Sources of interference are programs that are designed to exert pressure in shared resources. They usually implement some form of a *dial*, which is a mechanism that gradually increases the

Table 3.1: Characteristics of related work

Article	Resources	Objective	Domain
[6] Mars et al.	Memory	Prediction	Web services
[7] Delimitrou and Kozyrakis	CPU, memory, network, disk	Minimizing interference	Web services
[8] Govindan et al.	L3 cache	Prediction	Virtualized network functions
[9] Yang et al.	memory	Enforcing performance	Web services
[10] Manousis et al.	Memory, network	Prediction	Virtualized network functions
[11] Adeppady et al.	Memory, network	Prediction	Microservices
[12] Adeppady et al.	Memory, network	Prediction	Microservices
[21] Chen et al.	CPU, memory network, disk	Prediction	Co-resident virtual machines
[22] Huang et al.	CPU, memory network, disk	Prediction	Web services
[23] Rahman and Lama	CPU, memory network, disk	Prediction	Containerized microservices
[24] Mohammad et al.	Resource-agnostic	Prediction	IoT applications
[25] Kannan et al.	Resource-agnostic	Prediction	Co-located native workloads
[26] Breslow et al.	Resource-agnostic	Prediction	High-performance computing workloads
[19] Zhao et al.	CPU	Prediction	Multi-threaded applications

Table 3.2: Reviewed articles divided by approach

Article	Approach
[6] Mars et al. [7] Delimitrou and Kozyrakis [8] Govindan et al. [9] Yang et al.	Sources of Interference
[10] Manousis et al. [11] Adeppady et al. [12] Adeppady et al.	Regression with system metrics
[21] Chen et al.	Markov chain
[22] Huang et al. [23] Rahman and Lama	Neural network
[24] Mohammad et al.	Matrix facotrization
[25] Kannan et al. [26] Breslow et al. [19] Zhao et al.	Micro-experiments

SoI's contentiousness. They can be used along with profiling software to measure performance degradation caused by interference. This process is done in the following steps:

1. [**Measuring sensitivity**] A SoI is co-scheduled with an application running in isolation. Next, the pressure is gradually increased. By profiling the applications performance during varying interference, its sensitivity to interference can be quantified in the form of a *sensitivity curve*. This curve is a discrete function. It's domain is a set of dial levels $I : \{I_0, I_1, ..., I_{max}\}$, representing varying intensities of a SoI. The value of this function is the performance measured at given benchmark intensity.

2. [**Measuring contentiousness**] The contentiousness of other services is evaluated using a reporter—a lightweight program specifically designed to be highly sensitive to a particular type of resource contention. The reporter is co-located with a benchmark to derive its sensitivity curve. Subsequently, a workload is co-scheduled with the reporter. By analyzing the correlation between the observed performance degradation and the reporter's sensitivity curve, a benchmark intensity level I_C is identified. This intensity level applies roughly similar pressure on a given resource as the tested competitor, and is also known as its *contentiousness score*.

3. [**Predicting interference**] Sensitivity curves obtained from benchmark testing can serve as predictors of interference [6]. For a co-located pair of services: a target service A and its competitor C – interference can be estimated by plugging the SoI intensity I_C into the sensitivity curve of A. This allows for the prediction of performance degradation.

This approach seems to be very accurate, with average error for the of Bubble-Up methodology proposed by Mars et al. in the range of 1-2.2% [6]. However, Bubble-Up is limited to pairwise co-location. Yang et al. address this limitation with Bubble-Flux, which builds on the same core principles and enforces QoS by dynamically pausing and resuming multiple competing batch jobs [9]. Unfortunately, Bubble-Flux needs to actually deploy multiple competitors with the application under profiling in order to predict interference, which makes it unsuitable for our offline prediction purposes.

The scalability problem of Bubble-Up seems to be resolved by Govindan et al. using an optimization method known as *reduction tables* [8]. Govindan et al. prove that the aggregate interference of any pair of co-located SoI instances with dial settings $<I_1, I_2>$ can be mapped to an interference of a single SoI setting $I_3 = I_1 + I_2$, or $I_4 + I_{max} = I_1 + I_2$, where I_{max} denotes maximal intensity setting for a given SoI. The same reduction can also be applied for higher-dimensional tuples of co-located SoIs. This means that the contentiousness profiles of two or more co-located competitors can be combined to form a single profile. This method also exhibits promising accuracy; the authors report that the absolute prediction error is under 8% for almost 90% of analyzed workload combinations.

3.2.2 Regression With System Metrics

This approach was introduced by Manousis et al. [10] in the context of virtualized network functions, and later adopted for microservices by Adeppady et al. [11, 12]. It quantifies interference using system metrics obtained from tools like `mpstat` and Intel PCM. The interference prediction process contains three steps:

1. [**Sensitivity model training**] Each application is co-located with a tunable source of interference. Then, a *contentiousness vector* of the SoI is measured. It is a tuple of relevant system metrics $[m_0, m_1, ..., m_i]$. For example: $[m_0 = 100, m_1 = 20]$, where m_0 is LLC misses per second and m_1 is memory read traffic in GB/s. Next, the perfor-

mance of the application co-located with the competitor is measured and saved. This operation is repeated many times, with different intensities of the competitor, and the information is used to train *sensitivity model* $M_A : V \rightarrow P_A$, which is a regressor that can predict performance P_A of service A using an arbitrary contentiousness vector V. As opposed to the linear model of the sensitivity curve used in the SoI approach (described in Subsection 3.2.1) this is a multivariate model. In all three reviewed studies using the system metric regression approach, the model is implemented using the Gradient Boosting Regression.

2. [**Measuring contentiousness**] To obtain contentiousness vectors of real applications, they are also co-scheduled with a synthetic competitor. Then, system metrics are collected to form the contentiousness vector V_A of a an application A. This is also repeated, and an average contentiousness vector is created by combining all of the resulting vectors. The reason for scheduling the application with a synthetic competitor is that contentiousness measured in isolation would be inaccurate [10].

3. [**Predicting interference**] When considering a co-location of a target service A with multiple competitors $\{C_0, C_1, ..., C_n\}$, their contentiousness vectors $\{V_{C_0}, V_{C_1}, ..., V_{C_n}\}$ are combined into a single aggregate contentiousness vector V_C using linear operators appropriate for each system metric. Then, this vector is used with the model M_A to predict performance of the target service.

This approach has an average prediction error of less than 8% for Virtualized Network Functions (VNFs) [10]. While Adeppady et al. claim that they attain an error of 2.5-5.2% for microservices, it is worth to note that the workloads that they use (`snort`, `pkstat`, `MQTT`, and `nginx`) do not fall under the definition of microservices used in this thesis, as they are not components of a single system communicating with each other through the networks.

A notable advantage of this approach is its ability to accommodate more than two co-located competitors. However, our concern is the profiling time required for each application: Manousis et al. report that, to attain the desired accuracy, each VNF must be profiled across 1,000 distinct SoI configurations [10].

3.2.3 Markov Chains

This approach was presented by Chen et al.[27] to predict performance interference of Virtual Machines (VMs) running on the Xen [1] hypervisor. The method uses metrics exposed by the hypervisor monitoring tool `xentop` to model resource usage of a VM using Markov chains.

A Markov chain is a stochastic process that transitions between states according to fixed probabilities, where the probability of moving to the next state depends only on the current state. Chen et al. represent the processing steps of each competing workload as a Markov chain, with states corresponding to the currently accessed resource and transitions representing attempts to access the next resource. Their example model defines the state set $\{Dom0, pCPU, Disk, Net\}$, denoting the hypervisor control domain, physical CPU, disk, and network, respectively. Each state has two possible transitions, each occurring with a fixed probability within a given time interval: a self-loop, indicating a failed access attempt, and an outgoing transition, signifying successful resource acquisition. An example Markov chain can be seen on Figure 3.1.

Markov chains can be used to estimate the total time a workload spends on a resource, along with the slowdown caused by applications competing for the given resource [21]. When estimates of time spent on all resources are added together, the result is a prediction of the

[1] `https://xenproject.org/`

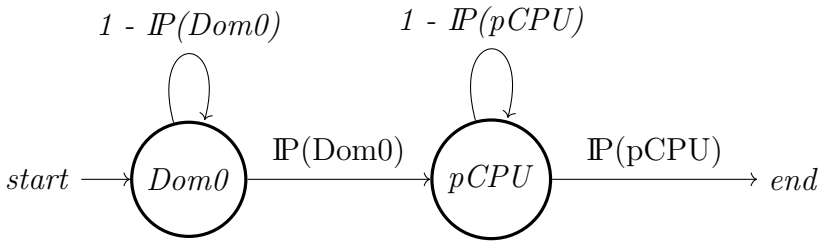


Figure 3.1: A Markov chain model of a CPU workload

total execution duration for the workload.

This approach offers prediction with a mean error of up to 20% [21]. Also, it assumes that the resource access patterns of the workloads do not change over time (for example due to user behavior), and the resource sharing policy and its underlying mechanism is known in advance (in the discussed example it is a M/G/1-PS queue).

3.2.4 Online Prediction With Neural Networks

This approach is featured in works of Rahman and Lama [23] and Huang et al. [22]. This technique also uses system metrics, collected using tools like `docker stats` and `virt top`. While conceptually similar to "regression with system metrics" approaches, it does not utilize synthetic workloads to generate contentiousness, and replaces simpler regression techniques with neural networks to hopefully capture more complex interactions between system metrics and performance interference. Instead of using all available metrics, which may cause overfitting, these studies employ stability selection, a feature selection technique designed to identify the most robust and informative predictors.

The method developed by Rahman and Lama has a mean absolute percentage error of up to 10% in certain cases [23]. The key feature that distinguishes their work from other approaches we discussed is its proven ability to predict performance degradation across the entire end-to-end system workflow.

A key drawback of these approaches in our context is that they require the application under profiling to be deployed alongside real competitors during metric collection, while our goal is to predict interference for components that have not previously been co-located.

3.2.5 Matrix Factorization

Mohammad et al. predict performance degradation using the matrix factorization technique [24]. They represent all of the possible application co-locations as a two-dimensional matrix of size $N \times M$, where N is the number of candidate applications, and M is the number of competitors (Table 3.3). Performance of some co-locations is measured by deploying applications pair-wise and profiling them. Next, Mohammad et al. employ the stochastic gradient descent method to estimate values for the remaining potential co-locations. This approach is based on the premise that a set of latent features can be learned by the algorithm from a limited sample set. For example, if a resource-intensive col-located application, C_1, degrades the performance of a candidate application, A_1, it is likely to have a similar impact on another candidate application, A_2. On the other hand, a non-resource-intensive competitor, C_2, is expected to have little to no effect on performance.

In order to generalize this method to co-locations of three applications, the matrix can be extended by replacing the set P of competitors $\{C_1, C_2, ..., C_M\}$ with a set of all possible combinations of co-located applications $\{<C_i, C_j> \in C^2 | i < j\}$. The method can also be generalized to higher-dimensional co-locations, but it is worth noting that the approach requires

Candidate applications \ Co-located applications	C_1	C_2	C_3	...	P_M
A_1	#	#	#	#	#
A_2	#				#
A_3	#				#
...	#				#
A_N	#	#	#	#	#

Table 3.3: *Matrix representation of co-locations*: The # symbols signify co-locations for which the performance was directly measured. The performance of the rest of the combinations needs to be estimated.

many deployments, because the complexity is $O(\binom{M}{N})$ where N is the number of co-located applications and M is the number of competitors. This may lead to a combinatorial explosion in execution time, and raises concern about the scalability of the method.

The authors report relatively high prediction accuracy: the approximation error does not exceed 8% for two co-located applications and remains below 14% for three. Nevertheless, the decrease in accuracy as the number of competitors grows is concerning, given our goal of eventually scaling the approach beyond pairwise co-locations.

3.2.6 Micro-Experiments

Existing research that employs micro-experiments to quantify interference includes the works of Kannan et al. [25], Breslow et al. [26], and Zhao et al. [19]. Micro-experiments temporarily pause or throttle applications for a few milliseconds to measure the performance degradation when running alongside competitors compared to running in isolation. However, because experiments are short-lived, detecting execution phases in the application and its co-runners is paramount to obtain accurate estimations of interference.

Kannan et al. [25] tackle the phase detection problem by utilizing performance monitoring units to identify the execution phases of applications, which the authors have empirically validated as effective for this task.

While the approaches of Kannan et al. [25] and Breslow et al. [26] are meant for online interference prediction between cloud applications, we believe they can also be used to estimate the performance of CPS product families. Since pausing applications is significantly faster than deploying them together, this approach could enable us to efficiently perform direct measurements of performance degradation. Kannan et al. report that this method achieves average error of 4%, even with 16 co-located applications.

In contrast to the previous two approaches, Zhao et al. [19] focus on predicting performance degradation specifically in the context of multi-threaded workloads sharing CPU cores. Their method exhibits prediction error in an approximate range of -10% to 10%, with one reported outlier of -32%. This approach is also able to profile workloads quickly, with one profiling run taking approximately 30 seconds. Additionally, the authors propose a set of SoIs capable of stealing CPU time from other applications. While they use them to study the causes of processor interference rather than for prediction, we believe they can be potentially reused by methods that utilize SoIs to profile sensitivity and contentiousness of applications.

3.3 Requirements

Even though performance interference is a relatively well-researched topic, it is mostly considered in the domain of cloud computing and network engineering. The dominant use-case for predicting resource contention is interference-aware application scheduling in data-centers. While this context is not entirely dissimilar to optimizing microservice-based CPS deployments, it presents a different set of unique challenges. In this section, we will discuss requirements specific to the problem at hand.

3.3.1 Analysis

We identify the following requirements that need to be met by a performance degradation prediction method to be deemed suitable in our context:

- [**Fit for prediction**] Reviewed approach should estimate performance degradation caused by resource contention with an accuracy of ±20%.
- [**Multiple competitors**] The method should be able to predict interference in co-location of up to ten microservices.
- [**Open source**] The code of used workloads and models, including sources of interference and contentiousness scoring mechanisms, should be publicly available. Alternatively, the paper should provide sufficiently detailed implementation descriptions to enable reproduction without additional research effort.
- [**Predictability vs. performance recommendations**] Modern multi-core systems include features that improve execution speed or hardware utilization but at the same time reduce performance predictability. Reviewed studies should provide practitioners with concrete recommendations for balancing this trade-off so that prediction accuracy remains acceptable without unnecessary concessions of performance and utilization.
- [**Appropriate for microservices**] The approach should accommodate microservice workloads.
- [**Fast profiling**] Our goal is to predict performance degradation of a product configuration that has not yet been integrated. Therefore, techniques that depend on large volumes of performance data from a fully operational system, require lengthy profiling, or need large numbers of services to be actually co-deployed, are impractical in our context.

To evaluate the potential applicability of existing work, we present a matrix in Table 3.4 that lists all the discussed articles along with the requirements they fulfill, may fulfill, or do not fulfill. The symbols that we used are explained in the Table 3.5.

Our analysis suggests that while some requirements are effectively addressed by the state of the art, no existing method fully satisfies all of them. Most of approaches provide accurate predictions of performance degradation and are capable of handling numerous co-located workloads. However, significant gaps remain, namely lack of directions for implementation of

Table 3.4: Requirements matrix

Articles	Accurate prediction	Multiple competitors	Open source	Predictability vs. performance recommendations	Specialized in microservices	Fast profiling
Mars et al. [6]	●		⊕			●
Delimitrou and Kozyrakis [7]		●	⊖			●
Govindan et al. [8]	●	●				●
Yang et al. [9]	●	●				
Manousis et al. [10]	●	●		⊖		⊖
Adeppady et al. [11]	●	●			●	⊖
Adeppady et al. [12]	●	●			●	⊖
Chen et al. [21]	⊖	●				●
Huang et al. [22]	●	●				
Rahman and Lama [23]	●	●			●	
Mohammad et al. [24]	●	●				⊖
Kannan et al. [25]	●	●				●
Breslow et al. [26]	⊖	●				●
Zhao et al. [19]	●		⊕			

Table 3.5: The meaning of symbols in the requirements matrix

Symbol	**Meaning**	**Description**
●	Requirement met	The author(s) of the work have explicitly demonstrated that the particular requirement is satisfied.
⊕	Requirement possibly met	Although the experiments do not conclusively demonstrate that the requirement is met, we consider the method to be promising in this particular aspect.
⊖	Requirement possibly not met	We suspect that the technique may not fulfill this particular requirement, although there is insufficient information to definitively rule out the possibility that it might.
	Requirement not met	The author(s) identified limitations that prevent the approach from fully meeting the requirement, or there is insufficient information to determine whether the requirement can be met.

synthetic workloads, and absence of practical guidance for obtaining reproducible, stable performance measurements. Notably, only three reviewed studies target microservices specifically, while approximately half of the surveyed approaches demonstrate potential for fast profiling.

3.3.2 Method Selection

Supported by our requirement analysis, we decide to extend the Bubble-Up methodology of Mars et al. [6] to fit our needs. We believe the disadvantages of the method outlined below are outweighed by it's advantages, and therefore it shows promise as a cornerstone for our research.

Disadvantages The Bubble-Up method does not scale beyond two competitors, and only models interference in the memory subsystem. It is also not specialized in microservice workloads, but in somewhat related domain monolithic web services and batch applications.

Advantages The method delivers accurate predictions with fast profiling times, and contains the most detailed recommendations on implementing the sources of interference and contentiousness scoring mechanisms. It also shows promise of multi-competitor scalability when supplemented with the approach of Govindan et al. [8].

3.3.3 Research Gap

While the Bubble-Up method [6] offers a promising foundation for a comprehensive performance-degradation prediction framework for validating CPS deployments, it does not satisfy several requirements of our study.

First, the method does not support more than two co-running applications. As mentioned in Section 1.2, we do not address this limitation, as we choose to leave it out of scope of this thesis. However, we believe that our literature study contains information that may be helpful in addressing this shortcoming.

Second, Mars et al. do not provide a complete, publicly available implementation of the contentiousness scoring mechanism; consequently, the approach fails to meet our "open source" requirement. We address this deficiency with our first contribution (C1), in which we propose an empirically validated design for the contentiousness scoring mechanism, and third contribution (C3), in which we release the full implementation of the method under an open-source license.

Third, Mars et al. provide no guidance on balancing prediction accuracy against runtime predictability, leaving the "predictability vs. performance" requirement unmet. We bridge this gap with our second contribution (C2), which consists of experimentally validated guidelines for configuring the execution environment.

Finally, the Bubble-Up method has not been validated on microservice workloads, so evidence that it satisfies the "specialized in microservices" requirement is inconclusive. We close this gap by validating the method on a corpus of microservice workloads, as described in Section 8.4.

Chapter 4

Predicting the Impact of Resource Contention

Selecting an appropriate method to predict interference-induced performance degradation is paramount for our research. In Chapter 3, we selected the Bubble-Up methodology proposed by Mars et al. [6], which models memory contention between two co-running applications. However, we also discovered that several crucial details of the approach are either not publicly disclosed or are insufficiently explained in the original publication.

This chapter outlines our approach to estimating application interference, adapted from the Bubble-Up methodology. First, we present a high-level overview of our method in Section 4.1. Next, we include detailed descriptions of sensitivity characterization (Section 4.2), and contentiousness profiling (Section 4.3). Finally, we discuss the prediction mechanism in Section 4.4.

4.1 Outline of the Method

We predict performance degradation caused interference between an application under analysis A and competing workload C using a multi-step approach. This method is illustrated in Figure 4.1, and consists of the following three steps:

1. In Step 1, we characterize *sensitivity* to interference of the application A. We achieve this by co-locating A with a *Source of Interference (SoI)*, which is a synthetic workload designed to induce variable amounts of contention in the memory subsystem. We then gradually increase the pressure generated by the SoI, and record the performance of A at every step. The result of this characterization is a sensitivity curve, which is a function mapping memory contention intensity (x-axis) to application's performance (y-axis).
2. In the Step 2, we measure *contentiousness* (i.e. the amount of pressure an application exerts on a shared resource) of the competitor C, by co-locating it with a *reporter* workload. The reporter is an application with a precomputed sensitivity curve that is responsive to memory interference and thus provides a common reference point. We then measure the reporter's performance when co-located with C, and obtain its *contentiousness score*, i.e. the intensity of memory pressure generated by C.
3. In Step 3, we predict the performance of application A by combining the two previously derived characteristics. Specifically, we evaluate A's sensitivity curve at the competitor's contentiousness score to obtain the predicted performance.

The process described above is able to predict performance degradation for one application, while the other is only characterized in terms of generated contention. Since our objective is

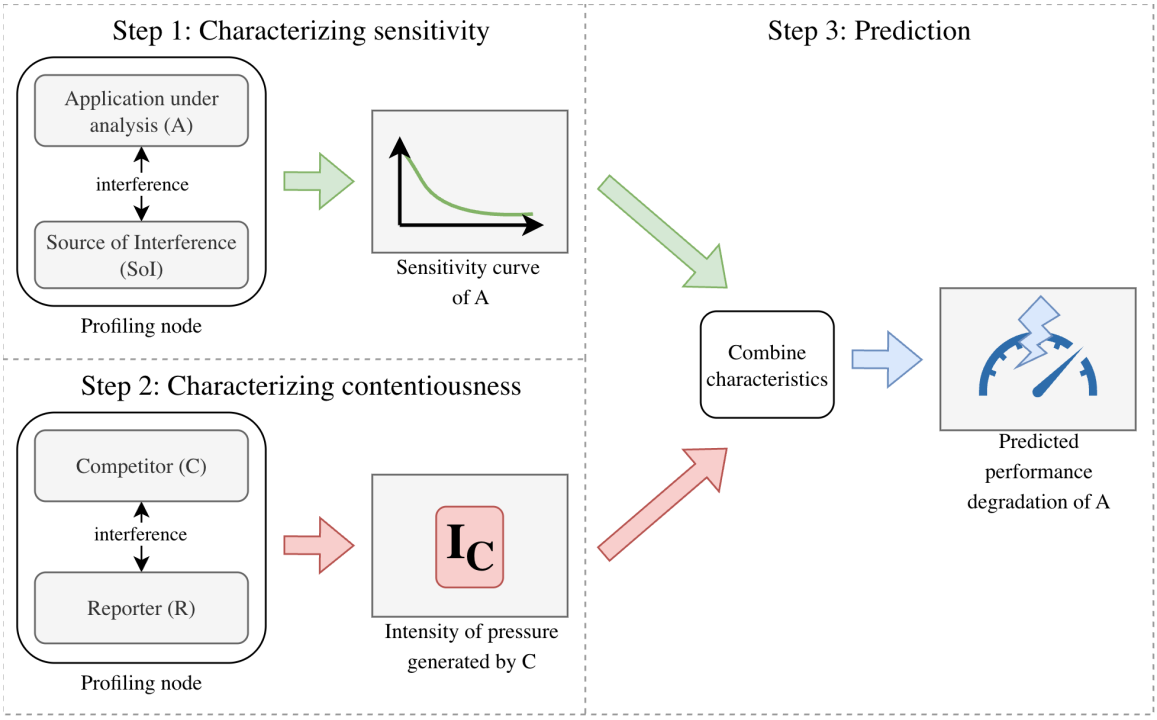


Figure 4.1: Overview of the 3 steps approach for predicting performance degradation

to predict the impact of *mutual* interference between two co-scheduled workloads W_1 and W_2, we need to repeat the 3-step prediction process twice, as illustrated in Figure 4.2: once with W_1 as the profiled application and W_2 as the competitor, and once with their roles reversed.

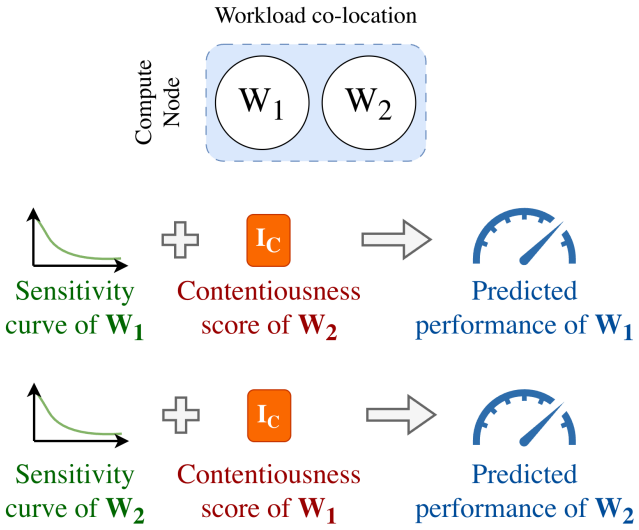


Figure 4.2: Predicting reciprocal performance degradation in a co-location of two workloads W_1 and W_2

4.2 Characterizing Sensitivity

We define an application's sensitivity to contention of a specific shared resource as the relationship between its performance and the intensity of interference affecting that resource. Applications may exhibit distinct sensitivity profile for different resources [7]. For the purposes

of this study, we restrict our analysis to the memory subsystem; however, we acknowledge that interference from other shared resources in the experimental setup may be present and could affect the accuracy of our results. Accordingly, all further references to *sensitivity* denote sensitivity to memory contention.

4.2.1 Sources of Interference

In order to measure the sensitivity of an application, we expose it to progressively increasing levels of resource contention in the memory subsystem. We achieve that using a Source of Interference (SoI). A SoI is a small contentious program designed specifically to synthesize pressure of variable intensity in a shared resource. The SoI can operate at a range of intensity settings $\{I_1, I_2, I_3, ...I_{max}\}$, where I_1 is an arbitrary small amount of pressure, and I_{max} is the maximum amount of pressure that can be generated. In an ideal case a SoI operating at I_{max} would fully saturate the shared resource, e.g. it would use the entire memory bandwidth or available disk I/O. However, for our method it is enough if the I_{max} setting generates at least as much interference as the most contentious application among the profiled workloads [6].

As we mentioned, our project is focused on predicting memory contention. Therefore, our SoI should produce pressure in the entire memory subsystem. We adopt the recommendations of Mars et al. [6] and create a multi-threaded program to serve as our SoI. This contentious kernel spawns two processes. The first process issues constant accesses in a streaming pattern, which stresses the cache prefetchers. A simplified example implementation of this pattern is shown in Listing 4.1; the `STREAM_SIZE` parameter on Line 1 controls the size of the working set, and thus the intensity of produced contention. The second process floods the memory with unpredictable accesses to pseudo-random addresses, which induce cache misses. The simplified implementation for this program is shown in Listing 4.2; its pressure can be changed by tweaking the `RAND_SIZE` parameter on Line 6. In order to maximize the contentiousness of both processes, we follow the guidelines of Mars et al. by increasing memory traffic through manual static single assignment (SSA) and employing a low-overhead linear feedback shift register (LFSR) pseudorandom number generator to minimize computation between random accesses.

```
1  #define STREAM_SIZE 8000000;
2  long bw_data_size = STREAM_SIZE * sizeof(double);
3
4  while(1) {
5      double *mid = bw_data + PADDING_SIZE;
6      #pragma omp parallel for
7      for (int i = 0; i < stream_size / 2; i += 10) {
8          bw_data[i]= scalar*mid[i];
9          bw_data[i+1]= scalar*mid[i+1];
10         ...
11         bw_data[i+9]= scalar*mid[i+9];
12     }
13     #pragma omp parallel for
14     for (int i = 0; i < stream_size / 2; i += 10) {
15         mid[i]= scalar*bw_data[i];
16         mid[i+1]= scalar*bw_data[i+1];
17         ...
18         mid[i+9]= scalar*bw_data[i+9];
19     }
20 }
```

Listing 4.1: Streaming access portion of the SoI

```
1  unsigned lfsr = 0xACE1u;
2  #define MASK 0xd0000001u
3  #define rand (lfsr = (lfsr >> 1) ^ (-(int)(lfsr & 1u) & MASK))
4  #define r (rand % rand_size)
5
6  #define RAND_SIZE 8000000
7  long rand_data_size = RAND_SIZE * sizeof(int64_t);
8  static volatile int64_t dump[200];
9
10 while(1) {
11     #pragma omp parallel for
12     for (int i = 0; i < 20; i++) {
13         dump[0]  += data_chunk[r]++;
14         dump[1]  += data_chunk[r]++;
15         ...
16         dump[199] += data_chunk[r]++;
17     }
18 }
```

Listing 4.2: Random access portion of the SoI

4.2.2 Sensitivity Curves

We assess sensitivity by co-running the application with progressively more aggressive SoIs, and benchmarking it's performance at every contention intensity level. We then graph the performance in relation to SoI intensity to create a *sensitivity curve*. We define sensitivity curve S_A of application A as a discrete set of ordered pairs:

$$S_A := \{(I_k, P_A(I_k)) | k = 0, 1, ..., max\} \tag{4.1}$$

Where $P_A(I_k)$ denotes performance of A measured when co-running with a SoI of intensity I_k. $P_A(I_0)$ is the definition of performance of A in isolation, i.e. when it is not co-scheduled with any other application or SoI.

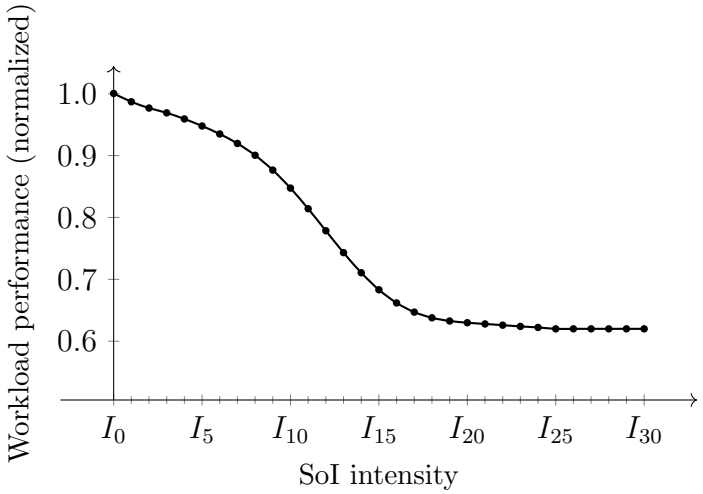


Figure 4.3: Example sensitivity curve of a workload co-located with a SoI.

We illustrate an example sensitvity curve on Figure 4.3. On the y-axis, we graph the performance measurements of the application, normalized to the performance in isolation. On the

x-axis, we show the SoI intensity settings. In the next steps of our approach, we will use the sensitivity curves to measure contentiousness, and predict the impact of interference.

Figure 4.4 visualizes the operating principles of the sensitivity profiling process in the context of an example memory hierarchy. The diagram represents an example machine architecture with four processor cores, each containing two hyperthread execution slots. The black arrows denote the pinning of applications to CPU cores. The application under profiling is deployed on the first two cores, and manipulates data that spills from the private caches into the L3 cache and DRAM memory, which are both shared between all workloads running on the compute node. At the same time, the SoI is deployed on the other two cores. The red arrows illustrate the dynamic nature of the pressure generated by the SoI: at the setting I_1, it utilizes some portion of the L3 cache and memory bandwidth, interfering with the application; at intensity I_2, it starts compete for a bigger portion of these components, possibly causing more performance degradation. At intensity I_3 the contention would be even greater. At every intensity $I_1, I_2, ..., I_{max}$, the performance of the application is recorded and added to its sensitivity curve.

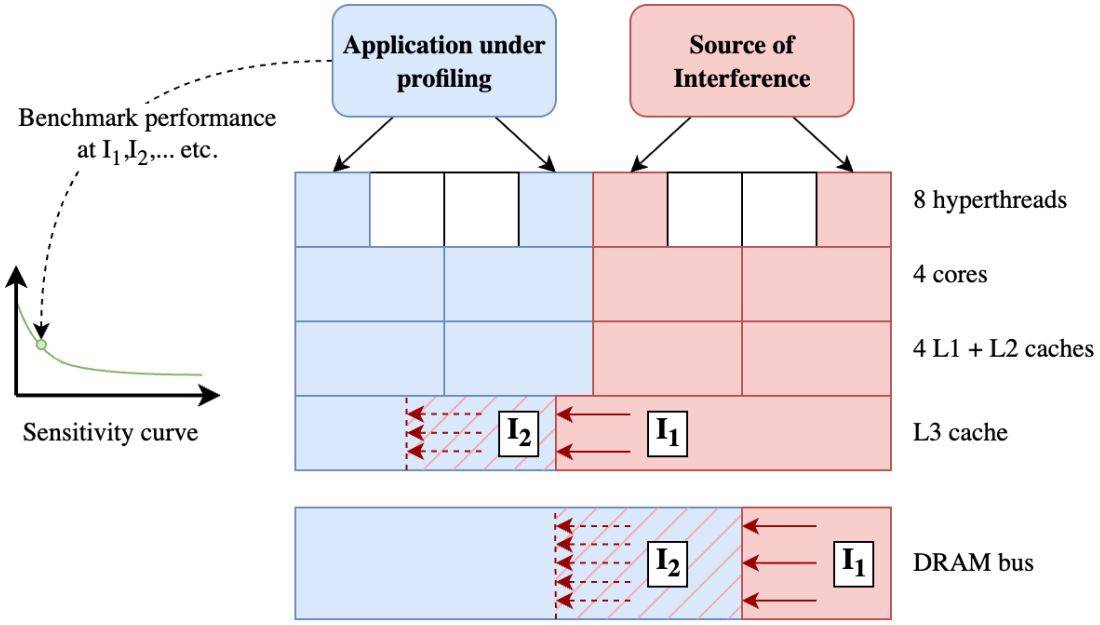


Figure 4.4: Characterizing the *sensitivity* of application under profiling, using a multi-threaded SoI with intensities $I_1, I_2, ..., I_{max}$.

4.3 Characterizing Contentiousness

Contentiousness of a workload can be coarsely approximated using performance counters, such as cache misses. However, this approach yields inaccurate results [28]. Instead, we approximate the contentiousness by running two workloads alongside each other:

- The *reporter* - a synthetic workload with a precomputed sensitivity curve. We further discuss our guidelines on designing a reporter in Section 4.3.
- The competitor, i.e the application which we want to profile in terms of contentiousness.

After co-locating the reporter R with a competitor C, we benchmark the performance of R, denoted as $P_R(C)$. This value allows us to relate the performance degradation caused by C to pressure generated by various intensities of the SoI. In order to relate the two, we use the following scoring mechanism: We find the closest pair $(I_k, P_R(I_k))$ from the discrete sensitivity curve S_R, such that $|P_R(C) - P_R(I_k)|$ is minimized. We then assign the *contentiousness score* of C to be equal to I_k. This means the interference generated by the SoI with intensity I_k is

representative of the competitor C. An example case of contentiousness scoring is illustrated in Figure 4.5.

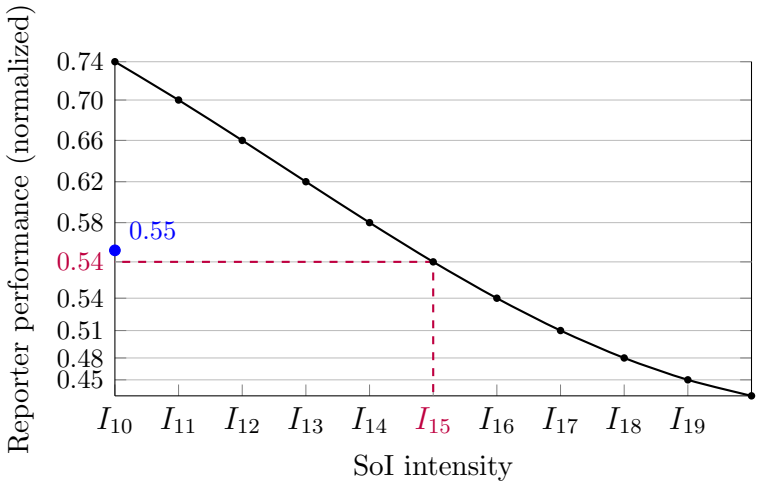


Figure 4.5: Example use of contention scoring mechanism. Given a measurement $P_R(C) = 0.55$ of reporter performance when co-located with competitor C (in blue), we find the closest reporter performance when profiled with the SoI (red). The corresponding SoI intensity (I_{15}) is the contentiousness score of the workload.

This process is further illustrated for an example memory subsystem in Figure 4.6. The memory hierarchy structure is the same as in Figure 4.4. Black arrows denote application-to-core pinning: in the example the reporter is pinned to a single core (since it is a single-threaded application), while a competing application occupies two cores. The pressure intensity, which is represented by the red arrows, remains static. The reporter's performance is profiled once, and the value produced by the previously described scoring operation is recorded as the competitor's contentiousness score.

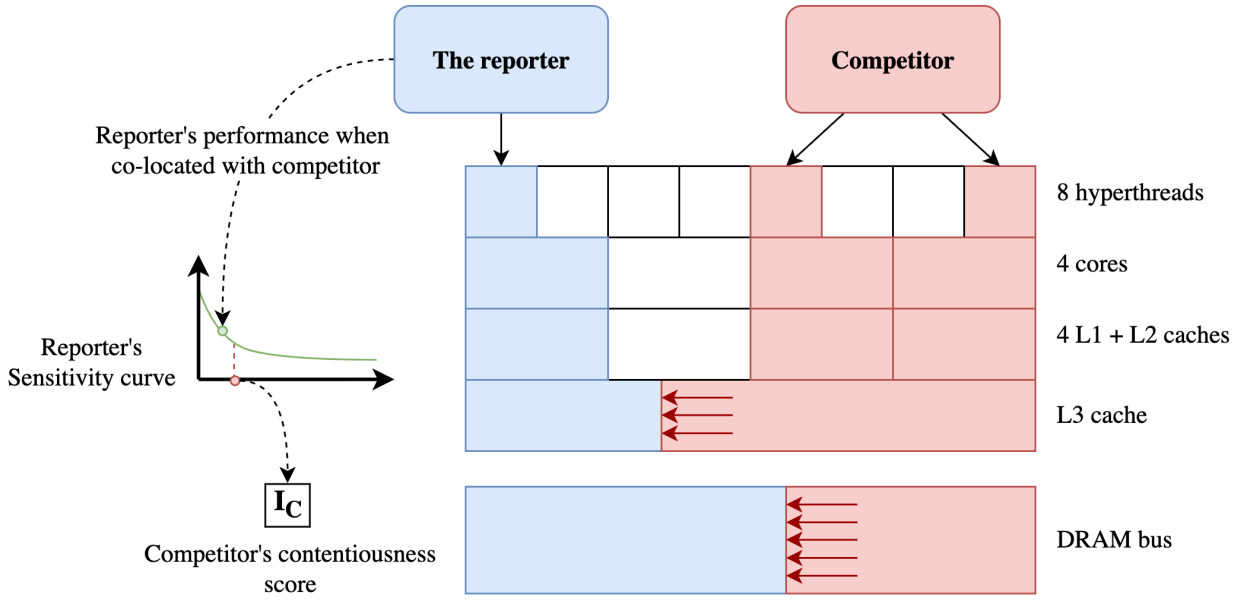


Figure 4.6: Characterizing the *contentiousness* of a competitor, using a single-threaded reporter workload.

4.4 Predicting Performance Degradation

After characterizing the *sensitivity* of an application under profiling A, and *contentiousness* of it's competitor C, we can combine these values in order to predict the resulting performance degradation. Given a sensitivity curve $S_A = \{(I_k, P_A(I_k))|k = 0, 1, ..., max\}$ of an application under analysis A, and competitor's C contentiousness score $I_C \in \{I_0, I_1, ..., I_{max}\}$, we define the predicted performance of A when co-located with C as:

$$\hat{P}_A(C) := S_A(I_C) \tag{4.2}$$

We demonstrate an example of this operation in Figure 4.7.

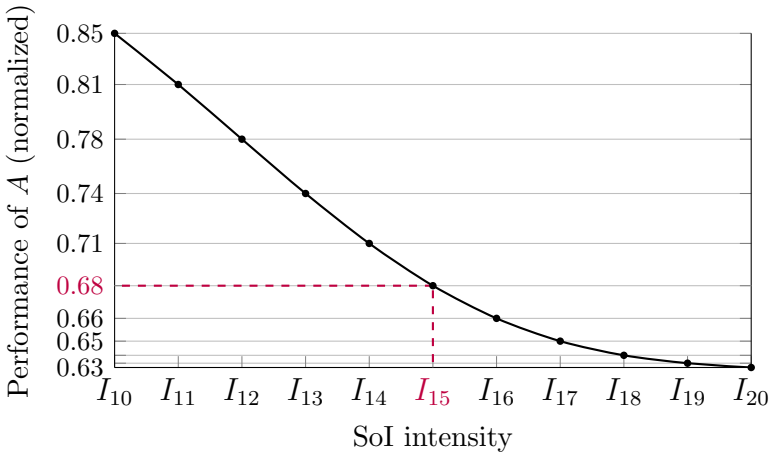


Figure 4.7: Example performance prediction of application A co-located with C. At C's contentiousness score I_{15} (in red), the sensitivity curve evaluates to 0.68, which is the predicted performance of A.

Chapter 5

Designing the Reporter

As we mentioned in Chapter 4, our method of predicting interference requires performing a contentiousness scoring operation. To achieve this, we leverage a reporter: a workload that is sensitive to contention in the shared resource and can be profiled to obtain a sensitivity curve. However, the original Bubble-Up methodology provides only general guidelines on reporter design and keeps the implementation closed-source [6]. A similar contention scoring approach is proposed in the iBench paper [7], where the authors re-use their sources of interference as reporters, but they also do not publish any details on how to turn these programs into contentiousness scoring mechanisms.

In this chapter, we aim to close this research gap by designing different variants of reporters, which constitutes our first novel contribution (C1). We begin by discussing broad design guidelines from literature in Section 5.1. Then, we introduce various memory access patterns in Section 5.2, and the reporter types in Section 5.3. Then, we discuss the impact of the working set size in Section 5.4, and share the implementation of an example reporter in Section 5.5.

5.1 Design Guidelines

Mars et al. [6] provide a single guideline for designing the reporter: it should be sensitive to the entire spectrum of possible memory subsystem contention, i.e. it should be impacted by pressure in the last-level cache, DRAM bandwidth, and the prefetcher. This means that: (1) the reporter should have a working set that is sufficiently large to not fit inside the CPU caches; and (2), the reporter should use a mix of streaming accesses to stress the prefetcher, and random accesses to produce cache misses. In order to validate those assumptions, we design four reporters using different memory access patterns. We also aim to release the code of the reporters under an open source license.

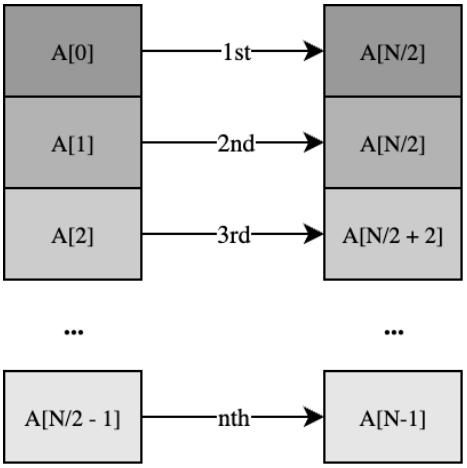
5.2 Memory Access Patterns

To stress the entire memory subsystem, we use three distinct access patterns: streaming, random, and hybrid. In the following subsections we describe their behavior and effects.

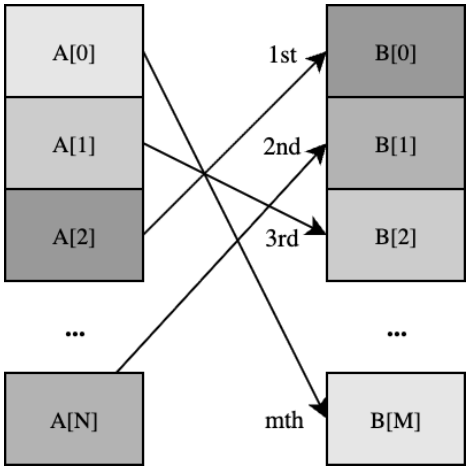
5.2.1 Streaming Access

This access pattern is based on the "copy" portion of the STREAM benchmark [13], and is similar to contention synthesis approaches proposed by Delimitrou and Kozyrakis [7] and Mars et al. [6]. The code sequentially traverses one half of a large array of double-precision floating-point numbers, reads each value, multiplies it by a scalar, and stores the result at the

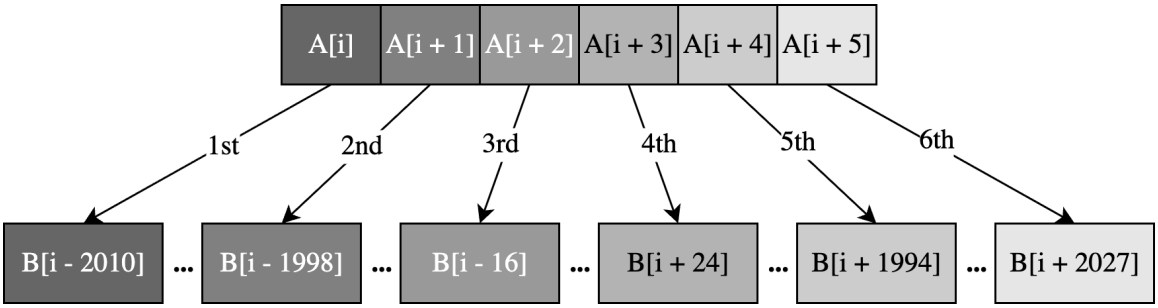
corresponding position in the second half of the array. as demonstrated in Figure 5.1a. The goal of using this pattern is to saturate the DRAM bandwidth while also triggering the cache prefetcher. This pattern is also utilized by the example SoI previously shown in Listing 4.1.



(a) Streaming memory access pattern: the code sequentially reads first $\frac{N}{2}$ elements of a large array A of size N, and stores them in the second half of the array.



(b) Random memory access pattern: elements from a large array A of size N are read in random order and stored at consecutive offsets in a smaller array B of size M.



(c) Hybrid memory access pattern: on each iteration a fixed-size block of consecutive elements from A (6 elements in the example) are read, and then written into B at predetermined, widely spaced offsets.

Figure 5.1: Visualization of memory access patterns

5.2.2 Random Access

While the streaming access pattern is sensitive to the memory bandwidth pressure, it yields relatively few cache misses due to memory prefetching. To induce cache misses, we propose a random access mechanism. It utilizes a large array of 64-bit integers, and a significantly smaller array of the same type. The code utilizes an LSFR pseudo-random number generator (also used by our SoI) to generate random offsets the range $[0, N]$, where N is the size of the large array. Then, we read data from the first array at those offsets and save it to the second array at consecutive indices. The second array is intentionally small (10 elements in our implementation), allowing it to fit entirely within the L1 cache. The random access portion of our example SoI (Listing 4.2) also uses this memory access scheme.

5.2.3 Hybrid Pattern

The third and final pattern uses a mix of streaming and "sparse" accesses. This pattern is based on the reverse engineering of the *lbm* benchmark from the SPEC CPU 2006 suite, conducted by Mars and Soffa [28]. We hypothesize that a reporter employing this pattern will both fill the caches by utilizing the prefetchers and saturate the memory bandwidth with semi-random writes. Therefore, such an application will have a comprehensive sensitivity to memory subsystem contention. The hybrid pattern is illustrated in Figure 5.1c.

5.3 Reporter Types

For our project we design four distinct types of reporters. All of them use a single-thread so that more CPU cores remain available to the co-located applications being profiled.

- **Types 1 and 2 -** These reporters each exercise a single memory access pattern (streaming and random, respectively). Because they produce only one kind of memory traffic, we use these reporters to test the hypothesis that an effective reporter needs to be sensitive to the memory subsystem as a whole.
- **Type 3 -** This reporter switches between the random and streaming access patterns. It is intended to be sensitive to the memory pressure holistically but may under-represent real applications that mix memory access types in a concurrent manner.
- **Type 4 -** The last reporter implements a hybrid pattern that issues streaming and random accesses one after another, and thus may better mirror the behavior of real-world workloads.

5.4 Working Set Size

As we mention in Chapter 5, a reporter should be sensitive to contention in all memory subsystem components, not just one. Moreover, the SoI should be able to achieve the same amount of pressure as the most contentious application in the profiling set, which may be impacting all levels of memory hierarchy, from the L1 cache of its core up to the DRAM bandwidth. This means that both the reporter and SoI should exercise the entirety of the memory subsystem [6]. To ensure that the sensitivity of the reporter extends well beyond the CPU caches, we configure its working set size to exceed the capacity of the last-level cache by a factor of eight.

5.5 Implementation

A crucial aspect of the reporter is the ability to accurately report its performance. Because of this, the reporter code should ideally leverage a proven microbenchmarking solution in order to profile it's execution and obtain desired performance metrics. For our implementation, we choose the Benchmark[1] library, an industry standard open-source framework for benchmarking code snippets. We implement our reporters in C++ because of its predictable memory management (e.g. the absence of a garbage collector) and compatibility with the Benchmark library. Since our project is mainly concerned with predicting latency-critical of workloads, we use the median execution time of the reporter code as its performance measure, rather than throughput or other metrics.

Listing 5.1 shows the implementation of the alternating-access reporter, while the implementations of the remaining reporter types appear in Appendix A. As we noted in Section 3.2.1,

[1] https://github.com/google/benchmark

the SoI used in our method employs both streaming and random access patterns and therefore produces memory traffic comparable to that generated by the alternating reporter. However, the alternating reporter and our example SoI are different in two substantial ways. First, the reporter must measure and accurately report its execution performance, necessitating the use of the Benchmark library; the SoI does not perform reporting and therefore does not require microbenchmarking. Second, the SoI is optimized to maximize potential contentiousness, whereas the reporter is optimized for sensitivity. In contrast to the SoI, which is multi-threaded, the reporter uses a single thread, which we hypothesize is easier to deprive of resources.

```
#include <benchmark/benchmark.h>

#define FOOTPRINT_SIZE 16000000
#define STREAM_SIZE (FOOTPRINT_SIZE / 2)
#define RAND_SIZE (FOOTPRINT_SIZE / 2)
#define PADDING_SIZE 400000
#define MASK 0xd0000001u
#define rand (lfsr = (lfsr >> 1) ^ (-(int)(lfsr & 1u) & MASK))
#define r (rand % RAND_SIZE)

unsigned lfsr;
static double bw_data [STREAM_SIZE + 2 * PADDING_SIZE];
static unsigned int data_chunk [RAND_SIZE];
static double scalar = 3.0;
static unsigned int dump[10];

void streaming_access(benchmark::State& state) {
    for (auto _ : state) {
        double *mid = bw_data + PADDING_SIZE;
        for (int i = 0; i < STREAM_SIZE / 2; i++) {
            bw_data[i]= scalar*mid[i];
        }
        for (int i = 0; i< STREAM_SIZE / 2; i++) {
            mid[i]= scalar*bw_data[i];
        }
    }
}

void random_access(benchmark::State& state) {
    for (auto _ : state) {
        lfsr = 0xACE1u;
        for (int i = 0; i < RAND_SIZE / 100; i++) {
            dump[0] += data_chunk[r]++;
            dump[1] += data_chunk[r]++;
            ...
            dump[9] += data_chunk[r]++;
            benchmark::ClobberMemory();
        }
    }
}

BENCHMARK(streaming_access);
BENCHMARK(random_access);
BENCHMARK_MAIN();
```

Listing 5.1: Our implementation of the alternating access reporter (Type 3)

Chapter 6

Improving Performance Stability in Contention Research

As we discussed in Chapter 4, accurate predictions using our method require that the performance of the workloads under analysis remain reasonably stable across profiling runs. In this chapter we present our recommendations for reducing inter-run performance variability, which constitute our second unique contribution (C2). We begin this chapter by examining the impact of performance instability on prediction accuracy in Section 6.1. We then identify two sources of preventable performance fluctuations, and propose mitigation strategies for each: execution speed variability caused by CPU frequency scaling mechanisms (Section 6.2) and inadequate workload isolation (Section 6.3).

6.1 Impact of Performance Variability

Each prediction of performance degradation relies on three independent measurements:

1. $P_A(I_C)$ - the performance of the application under analysis A when co-located with the source of interference configured to exert pressure with intensity I_C, which approximates the contentiousness of competing application C
2. $P_R(C)$ - the performance of the reporter R when co-located with the competitor C
3. $P_R(I_C)$ - the performance of the reporter R co-located with the SoI configured to operate at intensity I_C

Let us assume that the performance of profiled applications deviates between runs due variability. Let us also define P to be the baseline performance. In this case, each term of the equation calculating predicted performance $\hat{P}_A(C)$ will carry an associated term Pf, denoting the performance fluctuation attributed to variability:

$$\hat{P}_A(C) := P_A(I_C) + Pf_A(I_C), \tag{6.1}$$

$$\text{where} \quad I_C = \underset{I_k \in \{I_0, I_1, \ldots, I_{\max}\}}{\arg\min} |P_R(C) + Pf_R(C) - P_R(I_k) + Pf_R(I_k)| \tag{6.2}$$

Therefore, the variability in Equation (6.1) may cause application A to be characterized as either more or less sensitive than it truly is. Similarly, fluctuations appearing in Equation (6.2) can lead to incorrect estimation of the competitor's contentiousness (i.e., to selection of an inappropriate intensity I_C). In conclusion, prediction accuracy deteriorates unless performance is sufficiently stable across profiling runs.

Application performance can become unstable in two scenarios. First, if the application exhibits performance fluctuations due to varying input load or execution variability; in this

case, additional modeling of the application behavior is necessary, which is outside of the scope of this work. Second, when the performance changes because of external factors, including CPU frequency scaling mechanisms and unforeseen resource contention, e.g. due to the OS scheduling multiple workloads on the same CPU core. While some performance variability can be avoided by aggregating a large number of measurements [29], we recommend practitioners to take additional measures in order to remove the instability caused by external factors can be prevented.

6.2 Dynamic CPU Frequency Scaling

Dynamic CPU frequency-scaling mechanisms can amplify performance variability and consequently complicate benchmarking efforts [29, 30]. In this section, we describe their underlying mechanisms and briefly introduce our findings about their impact on prediction accuracy.

6.2.1 Scaling Governors

`CPUFreq` is a Linux kernel subsystem that allows the system or the user to change the CPU clock speed on the fly[1]. This feature is controlled by setting a CPU frequency governor, a kernel-level policy that either dynamically changes the CPU frequency or sets it to a constant value. As of the time of writing, the Linux kernel exposes six different governors, that we discuss in Table 6.1.

Table 6.1: CPU Frequency Governors

Governor	**Behavior**
Performance	Fixes the CPU frequency to the highest possible value.
Powersave	Fixes the CPU frequency to the lowest possible value.
Userspace	Allows any userspace program to fix frequency to an arbitrary value.
Ondemand	Dynamically varies the CPU frequency depending on the current system load.
Conservative	Operates similarly to the `ondemand` governor, but the frequency changes are more gradual.
Schedutil	Dynamically varies frequency by estimating load through the scheduler's Per-Entity Load Tracking (PELT) mechanism.

Kasture and Sanchez [30] propose `TailBench`, a framework for obtaining highly representative tail latency estimations. In their experimental setup, they set the CPU frequency to base (advertised) value by using the `userspace` governor. They do not provide empirical evidence for the positive impact of this configuration on performance variability, but their method is able to achieve desired end results. By contrast, Maricq et al. employ the `performance` governor for memory-subsystem stress testing and likewise obtain accurate performance estimates [29]. However, prior work has not examined how the choice of governor influences performance measurement in co-locations of multiple workloads. Governors which dynamically scale frequency can react to co-located workloads by substantially increasing clock speeds, potentially even producing environments in which an application performs better under co-location than in isolation. Because our method does not model the behavior of those mechanisms, we expect its predictions to be less accurate on systems with dynamic CPU frequency.

In order to make our methodology more flexible, we set out to investigate the impact of each governor on our methodology by performing experiments discussed in Section 8.2. We

[1] `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt)`

find out that using the `performance` governor provides best results with our method and hardware setup.

6.2.2 Turbo Boost

Turbo Boost[2] is a BIOS feature developed at Intel. It implements a CPU frequency scaling policy that is independent of the `CPUFreq` governor. Much like the `ondemand` governor, this feature overclocks the processor in a dynamic manner. It increases the frequency only when a core is under load and has sufficiently low temperature.[3]. Unlike `CPUFreq` governors, Turbo Boost may increase the frequency beyond kernel-defined maximum frequency boundary.

By disabling Turbo Boost in our experiments discussed in Section 8.3, we are able to reduce median absolute error in our prediction method by up to 10.5%. For that reason we advise against using this feature while predicting the impact of resource contention.

6.3 Isolating Workloads

Interference between workloads varies according to which memory subsystem components (i.e. CPU cores, caches and/or DRAM bus) are shared between the applications [18]. Therefore, applying our methodology requires knowledge about the memory-subsystem topology and the sizes of its caches. In the remainder of this section we develop a framework for reasoning about memory contention on a target architecture using *interference classes*, describe the procedure for configuring CPU affinity to ensure measurement validity, and specify how benchmark and reporter working sets should be sized relative to cache capacities.

6.3.1 Interference Classes

Not all co-resident applications on a chip exert the same pressure on one another. For example, two applications that share every level of cache will interact very differently from two applications that only share the DRAM bus. Therefore, categorizing workload co-locations according to shared memory subsystem components allows us to establish an equivalence relation between samples originating from different experiments on multi-core machines [18, 20]. To capture these distinctions, Kambadur et al. [18] formalize interference relationships between a pair of workloads using three interference classes (further illustrated in Figure 6.1):

1. ***Shared core*** **-** The workloads in this class execute on two hyperthreads that run on the same physical core. The workloads share all components of the memory subsystem, including the core itself.
2. ***Shared socket*** **-** The workloads use separate physical cores to execute. Core-exclusive L1 and L2 caches are not shared. Contention occurs in the last-level cache and the DRAM bus.
3. ***Different socket*** **-** This class occurs only on NUMA architectures with multiple independent chips. When two application hyperthreads are scheduled on two separate chips in different sockets, they share only the main memory.

We recommend applying our methodology to co-deployments belonging in the shared socket interference class. This workload isolation practice avoids shared core interference, which would arise because of added contention in the integer and floating point processing units [7, 18, 19], that is not modeled by our approach.

[2] `https://www.intel.com/content/www/us/en/gaming/resources/turbo-boost.html`
[3] `https://www.kernel.org/doc/Documentation/cpu-freq/boost.txt`

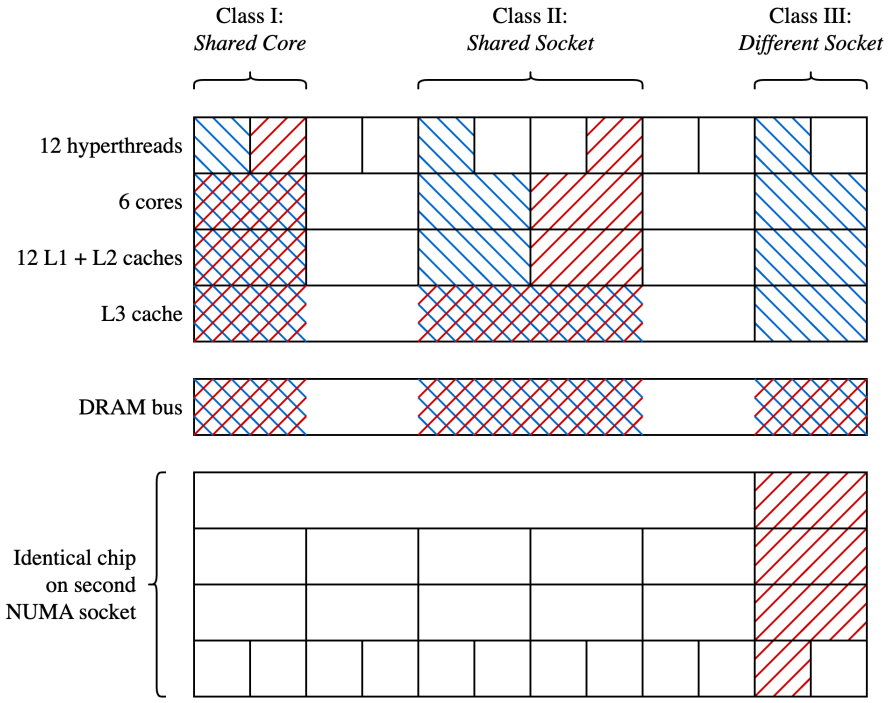


Figure 6.1: Interference classes according to [18]. The three classes are depicted using an example NUMA architecture with 6 cores and 12 hyperthreads.

6.3.2 CPU Affinity

In hyperthreaded architectures, the operating system exposes multiple logical cores, each acting as a hyperthread execution slot. In the example shown in Figure 6.1, the machine provides twelve logical cores (hyperthreads); each pair of logical cores is mapped to a single physical core (for example, logical cores 1 and 2 are mapped to physical core 1). Practitioners can configure each workload's processor affinity (this technique is also known as CPU pinning), effectively moving workloads between different interference classes. This can improve isolation by preventing interference that is not modeled by the prediction approach, i.e. the competition for the physical CPU core in case of our method. In the remainder of this section, we formalize the CPU pinning techniques to aid these configuration efforts.

On a machine with n logical cores, we define a co-location of two applications as two subsets A and B of the set of all logical cores $C = \{c_1, c_2, ...c_n\}$. Fore example, $A = \{c_1, c_2\}, B = \{c_3\}$ means that we have pinned application A to logical cores c_1 and c_2 and B to core c_3. We additionally define a shared core relation R on C, where $R(c_i, c_j)$ iff both c_i and c_j reside on the same physical core, and a shared socket relation S on C, where $S(c_i, c_j)$ iff c_i and c_j share a NUMA socket.

All co-locations for which the following is true belong to the **shared core** class:

$$\exists(c_a \in A, c_b \in B), \text{ such that } (c_a, c_b) \in R$$

All co-locations for which the following is true belong to the **shared socket** class:

$$\forall c_a \in A,\ \forall c_b \in B,\ \big((c_a, c_b) \notin R \wedge (c_a, c_b) \in S\big)$$

All other co-locations belong to the **different socket** class.

Chapter 7

Experimental setup

In this chapter we present our third contribution (C3): an experimental setup and the procedures used to evaluate our methodology and contributions discussed in prior chapters. We first give an overview of the high-level experimental approach in Section 7.1. We then describe the technical design of a testbed that facilitates experimentation in Section 7.2. Finally, we report the hardware and software configuration employed in our testbed in Section 7.3.

7.1 High Level Approach

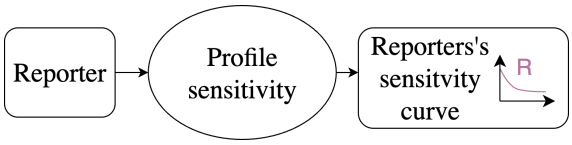
We validate our methodology by predicting performance of pair of workloads belonging to two distinct application corpora, and then validating the prediction accuracy. We also perform this experiment across multiple reporter and CPU frequency scaling configurations. The high-level approach is independent of the specific workload corpus and system configuration, and includes three steps, which we further illustrate in Figure 7.1:

1. In the first step (Fig. 7.1a), we train our reporter (i.e. profile its sensitivity) on the target architecture and configuration. The obtained reporter sensitivity curve will be used in the step 2 for contentiousness scoring.
2. In the second step (Fig. 7.1b), we profile each workload in the corpus in order to assess it's sensitivity and and contentiousness. These measurements provide the input data for prediction in step 3.
3. In the third step (Fig. 7.1c), we predict performance of each pair of workloads (A, B) in the corpus. First, we retrieve the sensitivity and contentiousness data obtained in step 2 to form a prediction. At the same time we co-locate A with B in order to measure the actual performance. Finally we validate the prediction by calculating the difference between both values, which is the error value.

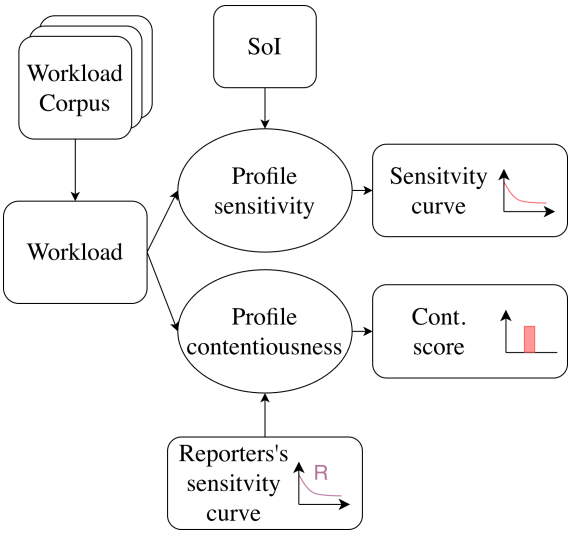
This procedure produces $|W|^2$ validated predictions, where $|W|$ denotes the cardinality of the workload corpus W. To assess the accuracy of our method, we examine each ordered co-location $(A, B) \in W \times W$ and define the prediction error for application A when co-located with B as

$$\varepsilon P_A(B) = \hat{P}_A(B) - P_A(B), \tag{7.1}$$

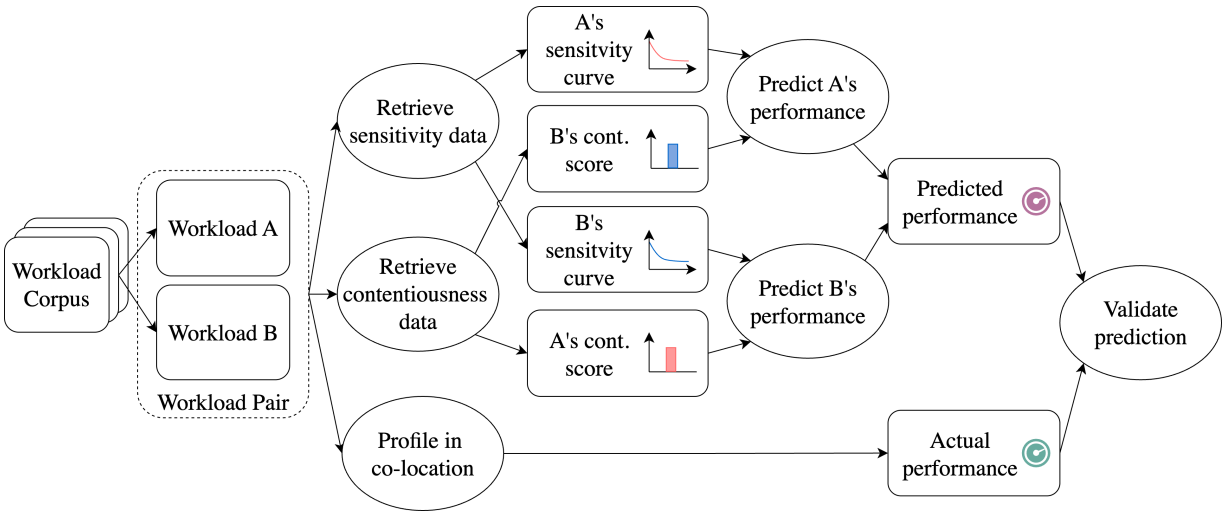
where $\hat{P}_A(B)$ denotes the predicted performance of A in the presence of B and $P_A(B)$ denotes the corresponding measured performance. We also compute the analogous error $\varepsilon P_B(A)$. Negative values of εP correspond to performance underestimation (the application executes faster than predicted), whereas positive values indicate that performance was overestimated by the method.



(a) Step 1 - reporter training



(b) Step 2 - workload profiling



(c) Step 3 - pairwise validation

Figure 7.1: The experimental approach broken down into steps.

7.2 Technical Design

In this section, we detail the technical design of a testbed used to perform experiments following the high-level approach.

7.2.1 Executors

Evaluation of our methodology requires deploying the SoI, applications and the reporter in various co-location scenarios and collecting performance measurements. To support this use case, we design *executors*: software modules responsible for launching and terminating workloads and for obtaining performance data where required.

SoI Executor The SoI Executor compiles and runs the SoI as a Linux process while allowing the client code to specify CPU affinity (to properly isolate the SoI from other workloads) and the interference intensity. There is no need to profile the performance of the SoI, so this executor lacks such functionality.

Workload Executor This executor's purpose is to deploy both the applications of interest and the reporter. We aim to keep our method agnostic of the execution platform. Because of this, the Workload Executor implements an *Abstract Executor* interface and delegates deployment and profiling responsibilities to a platform-specific *Concrete Executor* selected according to the workload's target execution environment. For example, when provided with a `yaml` manifest describing a Kubernetes deployment, the Abstract Executor delegates further operations to a Kubernetes Executor, which applies the manifest to the cluster and retrieves performance metrics from the database. We illustrate the principle of operation of the Workload Executor in Figure 7.2.

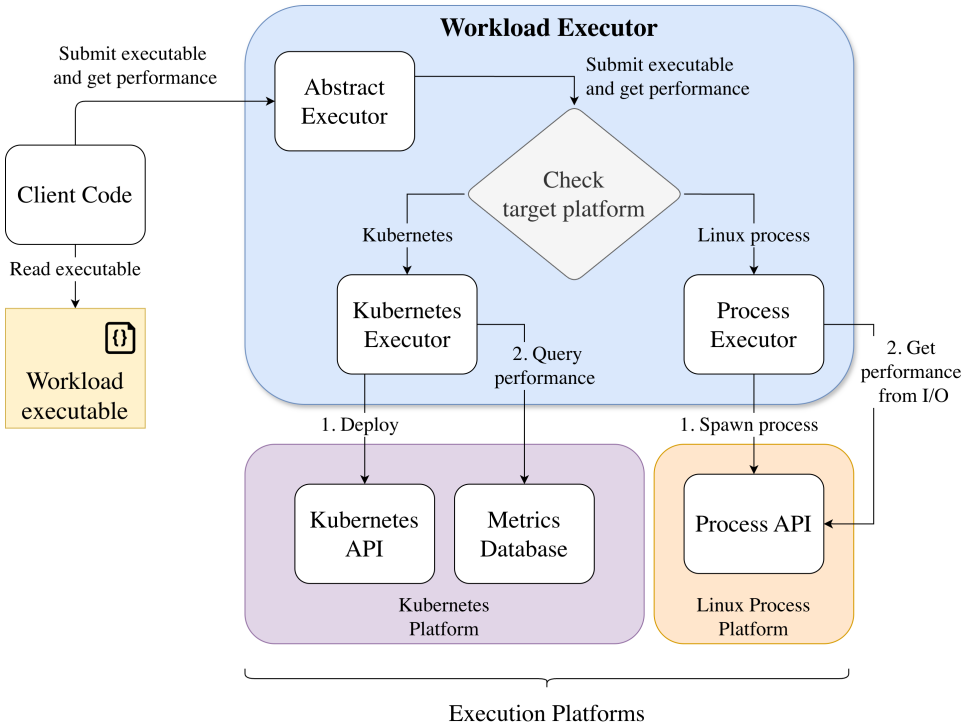


Figure 7.2: Architecture diagram of the Workload Executor

For our experiments we implement two concrete execution mediums: native Linux processes and Kubernetes objects. However, we aim for our design to be extensible, so that it can be

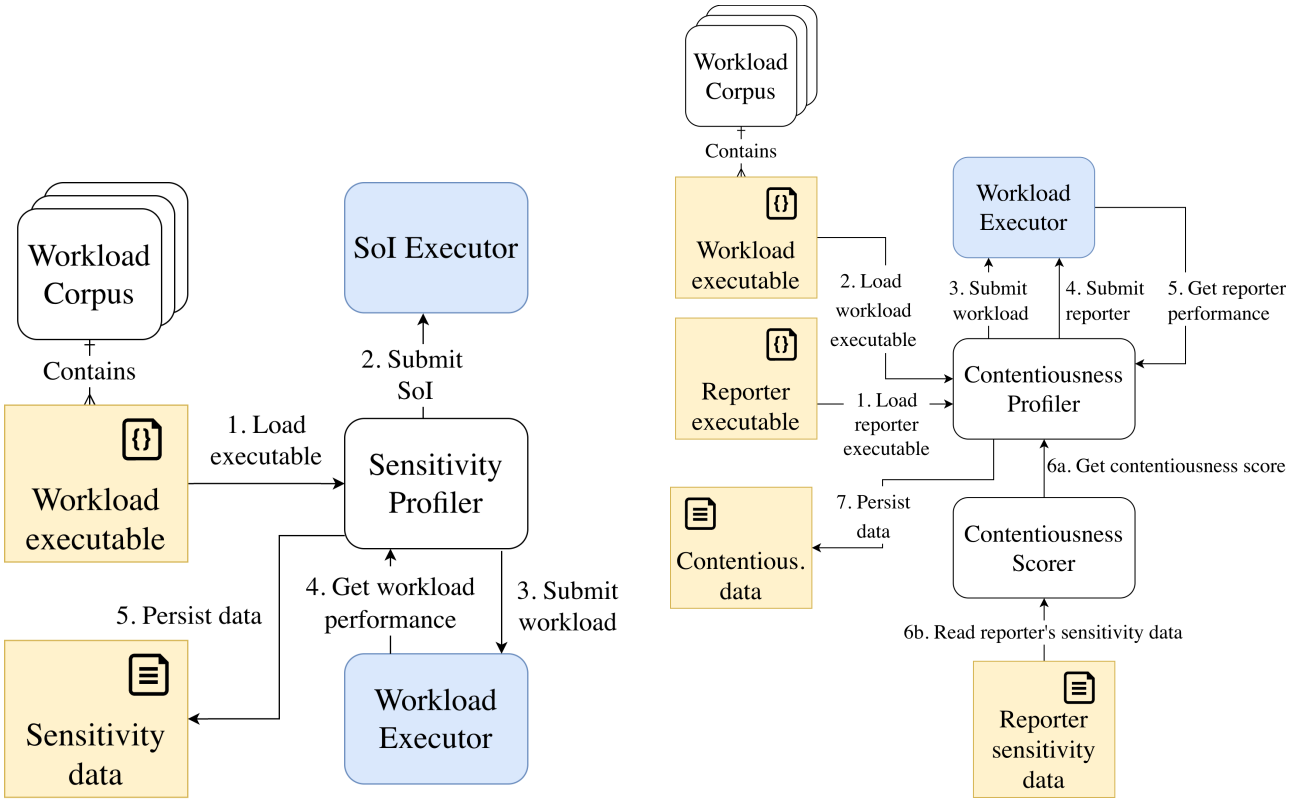
augmented with additional targets (e.g., Docker containers) if required.

7.2.2 Reporter Training

Reporter training uses the same procedure as sensitivity profiling for application workloads. Consequently, the reporter is profiled by the same sensitivity profiling flow described in the following subsection; the resulting sensitivity curve is then stored and used during contentiousness scoring.

7.2.3 Workload Profiling

The second step of the high-level approach can be split into two distinct flows: *sensitivity profiling* and *contentiousness profiling*. Both profiling flows are repeated for each individual workload in the corpus.



(a) Sensitivity profiling **(b) Contentiousness profiling**

Figure 7.3: Architecture diagram of application profiling

Sensitivity profiling This flow is controlled by the Sensitivity Profiler module, which performs the following steps for each workload in the corpus, and each SoI intensity I_k in the discrete range $[I_0, I_{max}]$:

1. Load the workload executable.
2. Submit the SoI with intensity I_k to the SoI Executor.
3. Deploy the workload using the Workload Executor.
4. Retrieve the workload's performance from the Workload Executor.
5. Append the sample pair $(I_k, P(I_k))$ to a CSV file that records the workload's sensitivity curve.

A detailed diagram of this flow is shown in Figure 7.3a.

Contentiousness profiling Contentiousness profiling is controlled by the Contentiousness Profiler. For each workload the following steps are executed:

1. Load the reporter executable.
2. Load the workload executable.
3. Deploy the workload using the Workload Executor.
4. Deploy the reporter using the Workload Executor.
5. Retrieve the reporter's performance data.
6. Compute the contentiousness score:
   (a) Provide the reporter performance to the Contentiousness Scorer.
   (b) Map the reporter performance onto the corresponding SoI intensity using the reporter's sensitivity curve.
7. Append the contentiousness score to a CSV file.

This procedure is illustrated in Figure 7.3b.

7.2.4 Prediction and Validation

Prediction and validation are implemented by two cooperating modules: the Predictor and the Validator. For each ordered pair $(A, B) \in W$ the following algorithm is executed:

1. Retrieve sensitivity curves for A and B.
2. Retrieve contentiousness scores for A and B.
3. Use the retrieved data to predict the performance of both applications in co-location (A, B) (using Equation 4.2), and append the predictions to a CSV file.
4. Read the predictions prepared for validation from the CSV file mentioned in step 3.
5. Load the workload executables for A and B.
6. Deploy both executables on the same machine using the Workload Executor.
7. Obtain measured performance of both workloads.
8. Compare predicted and measured performance to evaluate prediction accuracy (using Equation 7.1); persist validation results to a CSV file.

We summarize the prediction and validation flow in Figure 7.4.

7.3 System Configuration

Although we design our experiments to be architecture-agnostic, we report the hardware and operating system configuration to facilitate the reproducibility of our research. We also briefly describe the Kubernetes cluster used for our experiments.

7.3.1 Hardware

All experiments were executed on two identical machines, which satisfies our assumption that the compute nodes are homogeneous. Each machine was equipped with an 8-core Intel Xeon E-2378 processor. Every core featured private L1 data and instruction caches (384 KiB and 256 KiB, respectively) and a private 4 MiB L2 cache; a 16 MiB L3 cache was shared across all cores. The systems were provisioned with 64 GB of DRAM, with maximum bandwidth of 27.8

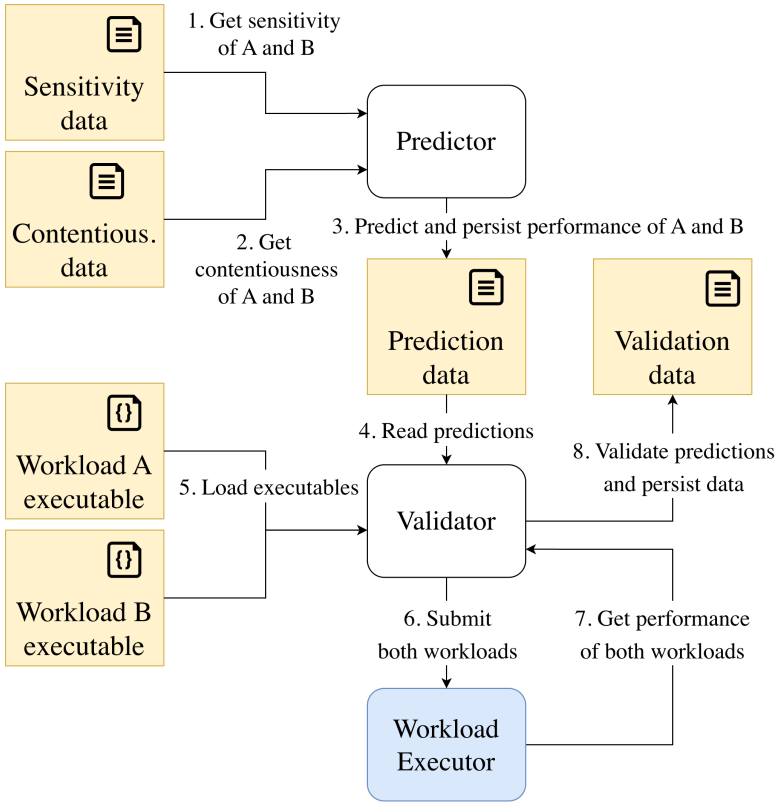


Figure 7.4: Architecture diagram of prediction and validation flows

GB/s, as reported by the "copy" portion of the STREAM benchmark[1]. The machines were interconnected via a 1 Gbps Ethernet link.

7.3.2 Operating System

Each experiment was conducted on Debian GNU/Linux 12 "Bookworm" using the default configuration. The only modifications applied were temporary adjustments to kernel parameters related to CPU frequency scaling for specific experiments, and are detailed in their respective sections.

7.3.3 Kubernetes Cluster

In order to validate predictions for Kubernetes workloads, we provisioned a two-node `MicroK8s` cluster[2] composed of the two experimental machines. One node was reserved exclusively for profiling and hosted only the minimal set of Kubernetes objects required for that purpose; the second node hosted a Prometheus Operator deployment[3] used to collect and query performance metrics and also housed any additional microservices that were part of the system under test but not actively profiled.

[1] `https://www.cs.virginia.edu/stream/ref.html`
[2] `https://microk8s.io/`
[3] `https://prometheus-operator.dev/`

Chapter 8

Evaluation

In this chapter, we present the results of utilizing the testbed introduced in Chapter 7 to conduct experiments on two corpora of applications: a suite of CPU-intensive benchmarks; and a Kubernetes-based system composed of microservices. First, we discuss the results for the benchmark case study in Section 8.1. Next, we examine how different reporters affect the accuracy of our methodology through sub-experiments in Section 8.2. We perform a similar analysis with various CPU frequency scaling mechanisms in Section 8.3. We conclude with a discussion of the microservice-based case study in Section 8.4.

The case study experiments are performed with type 3 reporter, the `performance` CPU scaling governor, and disabled Turbo Boost. We choose to showcase these results, as this configuration offers the most accurate prediction, as we discuss in Sections 8.2 and 8.3.

8.1 SPEC CPU Case Study

SPEC CPU 2017[1] is an industry-standard suite of computation-intensive benchmarks. These benchmarks include workloads such as: `mcf`, a bus route planning algorithm; `omnetpp`, a workload simulating large Ethernet networks; and `lbm`, a fluid dynamics simulator that is proven to induce an especially large amount of memory interference [28].

We chose SPEC CPU 2017 to validate the accuracy of our method, because its benchmarks represent a wide range of real-world use cases. Moreover, the suite and its predecessors have been widely used in prior studies on resource contention [7–9, 28], providing a valuable reference point for comparison with other methods.

8.1.1 The Corpus

The SPEC CPU 2017 suite contains two versions of each benchmark: `SPECspeed`, which reports execution duration; and `SPECrate`, which reports the throughput. Due to our focus on latency-sensitive applications, we only use workloads from `SPECspeed` category to validate our method. We use 19 benchmarks from the `SPECspeed` suite, while excluding `wrf`, due to compilation issues.

SPEC benchmarks provide three input sizes, in ascending order: `test`, `train`, and `ref`[2]. Although official SPEC submissions require running each workload with all three input datasets, executing the workload at the `ref` size was prohibitively time-consuming on our testing infrastructure. For example, a single `lbm` profiling run required 625 seconds; this needs to be repeated with 30 SoI intensity settings, and the whole process for one benchmark would therefore take ≈ 5 hours. Since our suite contains 19 benchmarks, performing the same

[1] `https://www.spec.org/cpu2017/Docs/overview.html#benchmarks`

[2] `https://www.spec.org/cpu2017/docs/runcpu.html#size`

procedure across the suite would take several days, making experimentation at the `ref` size impractical for this study. Therefore, we conducted our experiments using the `train` input exclusively; we consider this size sufficient to validate the proposed methodology.

8.1.2 Profiling Results

Figure 8.1 presents the sensitivity curves of 19 SPEC CPU benchmarks used in our study. The x-axis of each chart represents the SoI intensity setting in MB, while the y-axis represents the normalized performance of the benchmark.

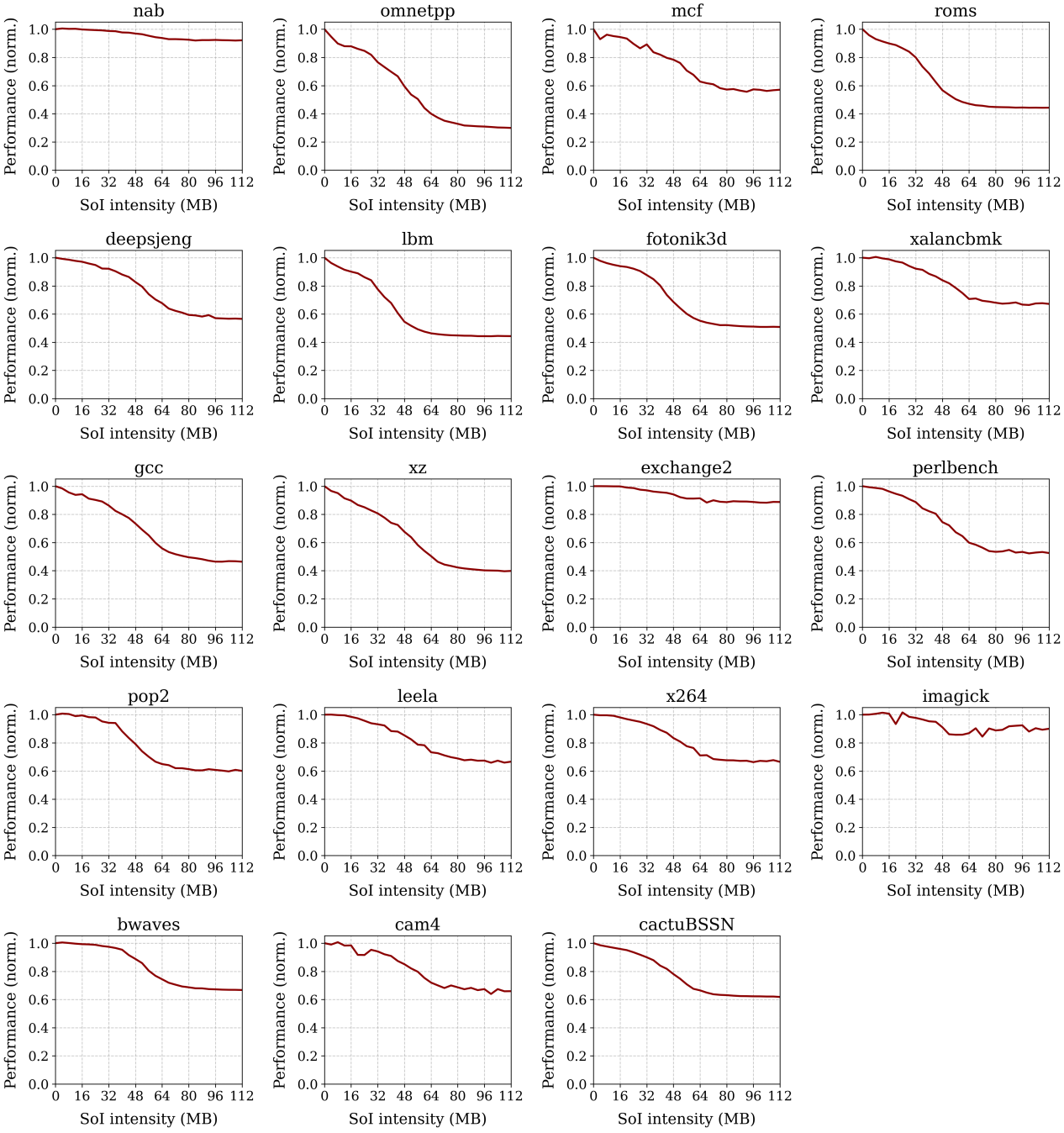


Figure 8.1: Sensitivity curves of SPEC CPU benchmarks

We observe that `omnetpp` is the most sensitive of the corpus, suffering a performance degradation of 70% at $I_{max} = 112$. In contrast, `nab` benchmark is the least sensitive in the corpus, suffering only 7.9% degradation during maximum contention. All curves are mostly monotonic, with some irregularities that we attribute to system variability. These anomalies are especially visible on the sensitivity charts of `imagick` and `cam4`. Additionally, for SoI working sets larger than 16 MB (which is the size of our L3 cache, as discussed in Section 7.3), we observe a monotonic increase in performance degradation. This indicates that pressure spills into main-memory bandwidth [6], thereby causing contention across the entire memory subsystem.

In Figure 8.2, we graph the contentiousness scores of all benchmarks. The scores across our corpus vary from 0 MB (i.e. the amount of memory pressure generated is zero or negligible) to 60 MB for the most contentious benchmark, `lbm`. While the maximum possible intensity of our SoI is 112 MB, we notice that most sensitivity curves flatten when interference intensity reaches 64-96 MB, which means that `lbm` can saturate a large portion of the memory bandwidth. We also observe that, sensitivity does not seem to be correlated with contentiousness. For example, `nab` and `omnetpp` exhibit substantially greater performance degradation than `bwaves`, despite having contentiousness scores of only 0 MB and 8 MB, respectively; `bwaves` has a higher contentiousness score of 36 MB.

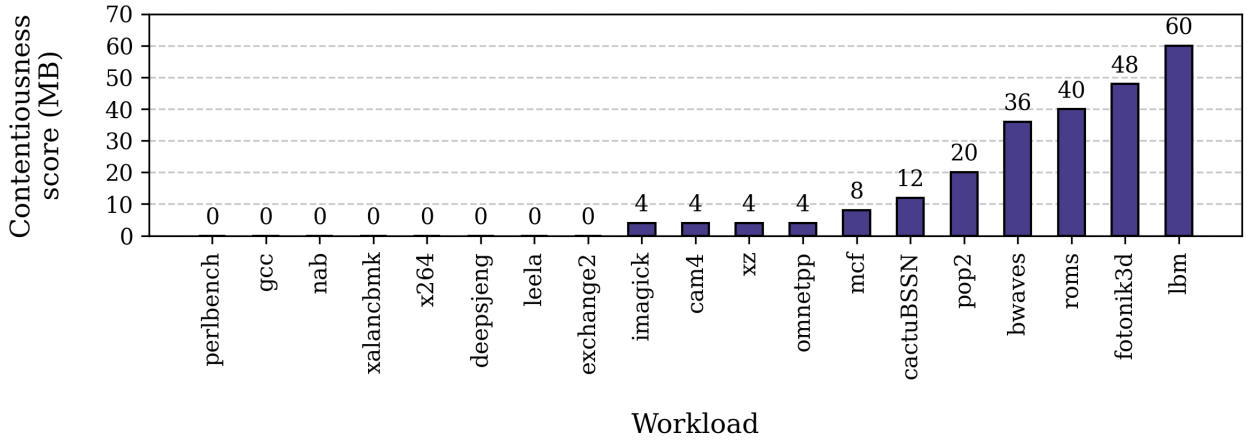


Figure 8.2: Contentiousness scores of SPEC CPU benchmarks

8.1.3 Prediction Results

To evaluate the prediction accuracy of our methodology, we perform step 3 of our experimental approach described in Section 7.1, and generate 342 predictions. Next, we validate the predicted performance values by co-locating each pair of workloads and measuring their actual performance.

We depict the overall distribution of error in Figure 8.3. The x-axis represents the prediction error, and the y-axis shows the frequency (count) of observations for each error value. The prediction error is calculated by subtracting actual performance from estimated performance (Equation 7.1); positive values indicate overestimation of performance, while negative values denote underestimation. The minimum and maximum recorded error is -10.4% and 16.2% respectively. The method exhibits low median absolute error of 0.98%, while 90th percentile stands at 5.5%, and the 99th at 10.4%.

Figure 8.4 presents the average absolute error of predictions (y-axis), clustered according to the competitor contentiousness score (x-axis). In experiments with competitors possessing contentiousness scores $\geq$ 36 MB, the average absolute error increases by a factor of two or more

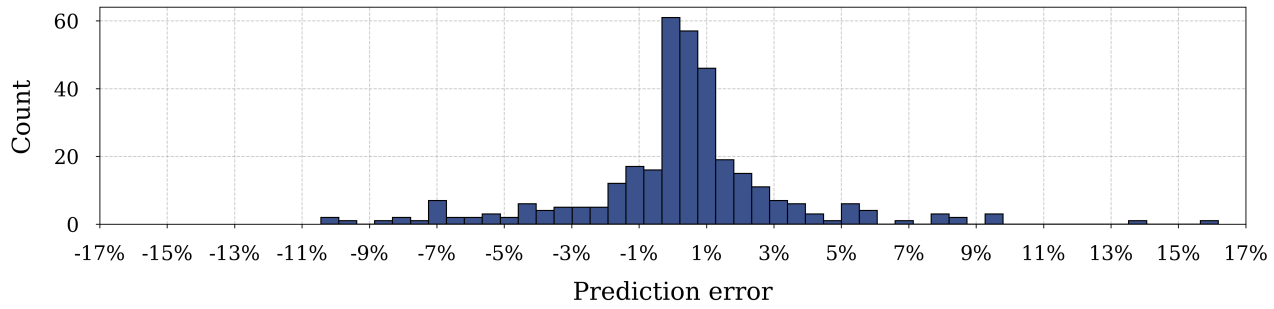


Figure 8.3: Error distribution of resource contention prediction

compared to experiments with less aggressive competitors. This indicates that the method may produce less accurate predictions in highly contentious co-locations.

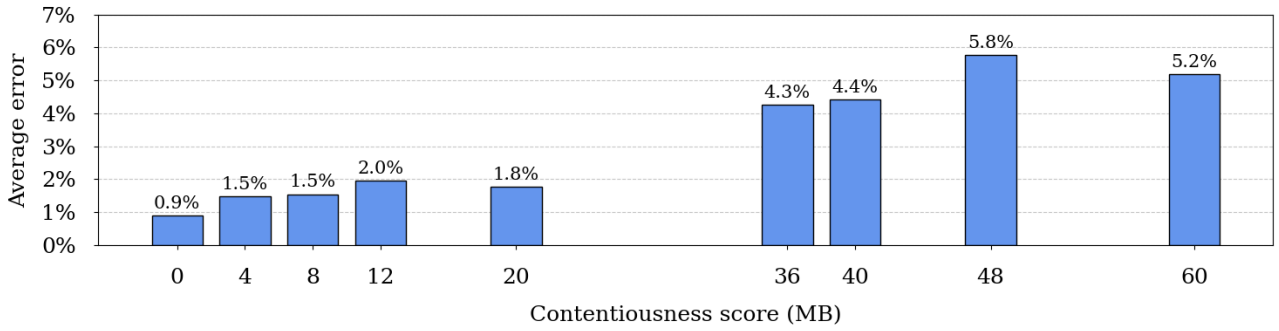


Figure 8.4: Average prediction error grouped by competitor's contentiousness

8.2 Reporter Evaluation

In this section we showcase the results of replicating our experiments on the SPEC CPU corpus using four distinct reporter types. Figure 8.5 shows the absolute error distribution for Types 1–4 (lower values are better).

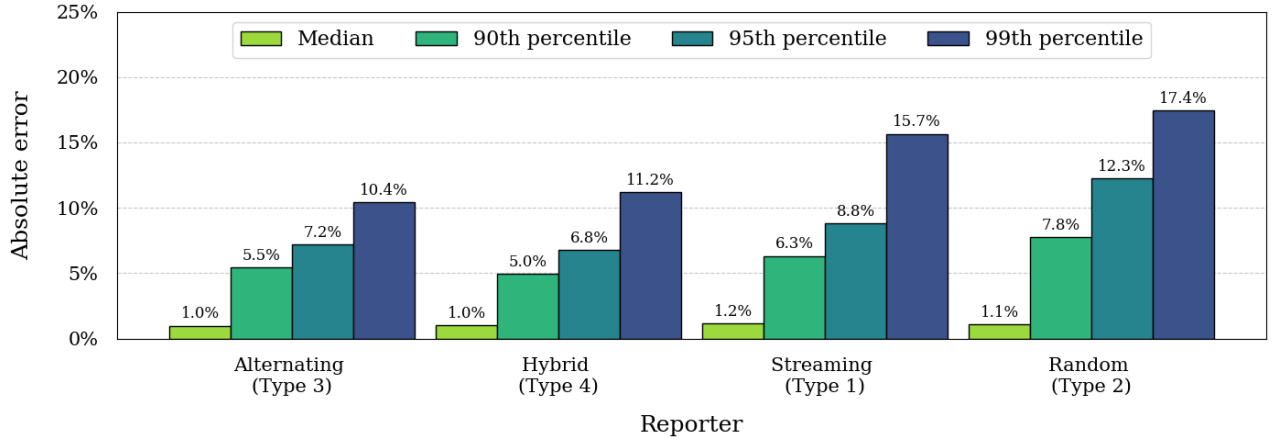


Figure 8.5: Error distribution by reporter type

Predictions made with the alternating (Type 3) and hybrid (Type 4) reporters are the most accurate, with similar error distributions. The streaming reporter (Type 1) is less accurate

than the aforementioned types, while the random reporter (Type 2) produces the largest errors. The results suggest that while the implementation of the reporter does not improve the median absolute error of our method by a large margin, it can improve 90th percentile absolute error by as much as 2.8%, 95th percentile by up to 5.5%, and 99th by up to 7%. With respect to selection an appropriate reporter, Type 4 achieves better accuracy than Type 3 in at least 95% of predictions, whereas Type 3 is slightly more effective at predicting extreme outliers. Accordingly, we recommend that practitioners employ the Type 4 reporter to attain the best accuracy in most scenarios, unless accurate prediction of the long tail (99th percentile) is critical, in which case reporter Type 3 may be able to attain better results.

We present the full results of accuracy validation with different reporter types in Appendix B.

8.3 CPU Frequency Scaling Evaluation

In this section we describe the results obtained by re-running our experimental approach on the SPEC CPU corpus while using different CPU frequency scaling mechanisms. Figure 8.6 illustrates the distribution of absolute error for all available `CPUFreq` governors, and Turbo Boost (lower values are better).

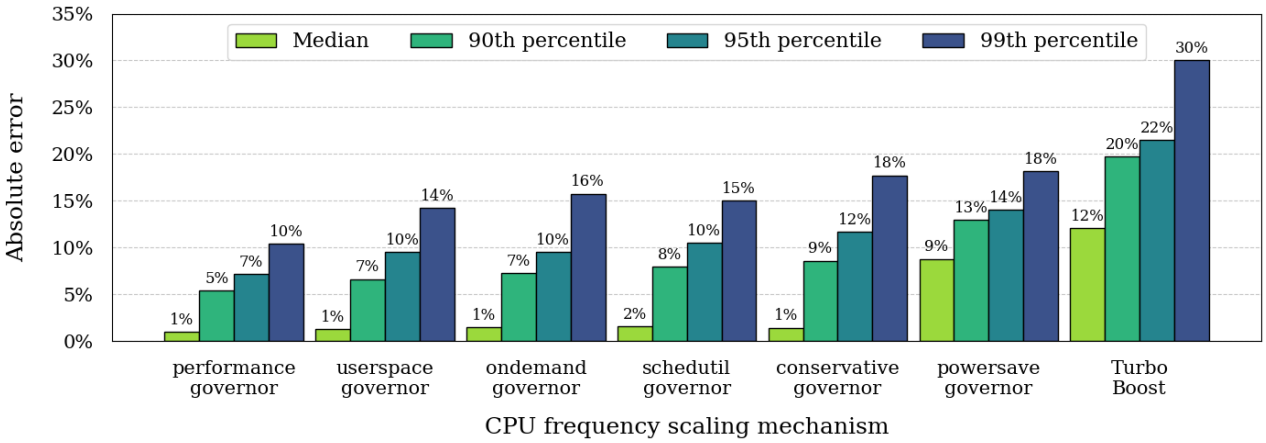


Figure 8.6: Error distribution by CPU frequency scaling mechanism

We attain the best accuracy when utilizing `performance` governor, with just 10% error in the 99th percentile, closely followed by the `userspace` governor; while the former keeps the frequency fixed at maximum possible value, we observe slight fluctuations (in the range of $[0.85; 1]$ of maximum frequency) during the experiments with the latter. We hypothesize these variations are caused by OS workloads interrupting applications under analysis and changing the CPU frequency. In contrast, the dynamic `ondemand`, `schedutil`, and `conservative` governors show larger errors, especially in higher percentiles. The last governor, `powersave`, performs substantially worse than other governors, despite using a fixed frequency for all processes. Finally, we observe the lowest accuracy when Turbo Boost is enabled while the CPU governor is set to `performance`; this configuration was chosen to avoid overlapping dynamic frequency-scaling mechanisms and thus better isolate Turbo Boost's effect on accuracy. We observe a general trend in our results, where dynamically varying CPU frequency leads to degradation in accuracy.

To investigate the trade-off between performance and predictability when selecting CPU frequency-scaling policies, we report the total duration of sensitivity profiling for 19 SPEC benchmarks under each scaling configuration (Table 8.1). Profiling durations for all configurations except the `powersave` governor vary only by ±9.9 minutes, which indicates that

performance gains among these settings are negligible. In contrast, the `powersave` governor incurs a substantial penalty: its profiling runs are roughly 9 hours longer than those of the other configurations.

Table 8.1: Time to complete sensitivity profiling

Configuration	Time (h)
performance governor	8.66
userspace governor	8.73
ondemand governor	8.81
schedutil governor	8.66
conservative governor	8.76
powersave governor	17.45
Turbo Boost	8.48

Our results suggest that practitioners should adopt the `performance` governor with Turbo Boost disabled to maximize profiling speed and prediction accuracy. Using the `powersave` governor results poor predictive accuracy and incurs a substantial runtime penalty, and thus should be avoided except when energy conservation is an overriding priority. Although enabling Turbo Boost produces a modest ($\approx 2\%$) improvement in profiling throughput relative to the `performance` governor, it substantially degrades prediction accuracy; consequently, we do not recommend using Turbo Boost when prediction accuracy is important.

We include complete results of evaluating the prediction accuracy with different CPU frequency scaling configuration in Appendix C.

8.4 Meal Delivery System Case Study

In the second case study, we set out to investigate the impact of inter-workload dependencies and network calls on the performance of our method. While SPEC benchmarks enable us to test the method with realistic workloads, each benchmark operates in isolation and does not communicate through the network with other workloads. Therefore, we do not claim that our method directly generalizes to the microservices domain based on the first case study. Instead, we evaluate it on a subset of three microservices from the Meal Delivery System (MDS). MDS is a benchmark system developed at TNO-ESI, that includes relevant features of a real-world CPS, and therefore we deem it more representative of our context than the SPEC CPU corpus.

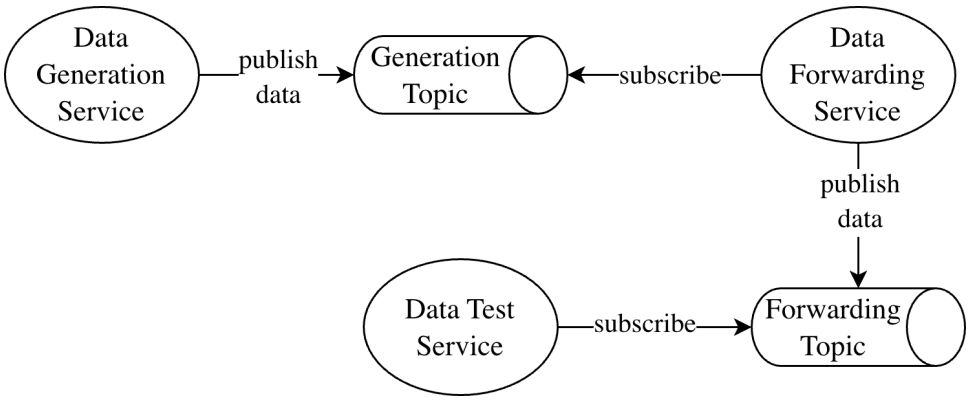


Figure 8.7: MDS system diagram

8.4.1 The Corpus

MDS is composed of 3 microservices that are interconnected using the publish-subscribe pattern of the ZeroMQ[3] networking library. The data flow of the system is depicted in Figure 8.7, and individual microservices are further described in Table 8.2. All of the services are single-threaded and process the data sequentially. The pressure exerted on the memory subsystem by MDS microservices is very low (they exhibit a maximum contentiousness score of 4 out of 112 MB). Consequently, the Meal Delivery System is not especially well suited to validate our method on its own. We aim to close this gap by extending the MDS workload corpus with SPEC benchmarks, which we will use as proxies for competitive neighbors. For this purpose, we selected the following benchmarks, identified as the most contentious in our previous case study: `cactuBSSN`, `pop2`, `bwaves`, `roms`, `fotonik3d`, and `lbm`.

Table 8.2: MDS microservices and their functionality

Name	**Functionality**	**Contentiousness Score (MB)**
Data Generation Service (DGS)	Generate blocks of 50000 pseudo-random 64-bit integers, and publish each block as a message to be consumed by the next service.	4/112
Data Forwarding Service (DFS)	Read data generated by the Data Generation Service and publish it to the next topic.	4/112
Data Test Service (DTS)	Read data forwarded by the Data Forwarding Service and perform the fast Fourier transform on the data.	0/112

8.4.2 Profiling Results

During the sensitivity profiling of the MDS microservices, we encountered a significant issue. Because these services process data sequentially in a single thread, the latency of the ZeroMQ network call used to transmit data accounts for approximately 90% of the total execution time of the profiled operation. In other words, the MDS microservices are much less sensitive to memory contention than network contention. Since our approach does not model network contention, we decided to exclude network latency from execution time measurements.

The resulting sensitivity curves for the MDS microservices are shown in Figure 8.8. Among the three, DFS exhibits the highest sensitivity to memory contention, while DGS is rather insensitive. Notably, DFS performs mixed read and write operations due to data copying, DTS performs only reads interspersed with data processing, and DGS focuses exclusively on writes. We believe that these results may signify that memory writes are less sensitive to contention because they are non-blocking.

Figure 8.9 illustrates the contentiousness scores of the MDS microservices alongside six SPEC benchmarks selected for comparison. Overall, the MDS microservices generate substantially less memory pressure than their SPEC counterparts.

[3] `https://zeromq.org/`

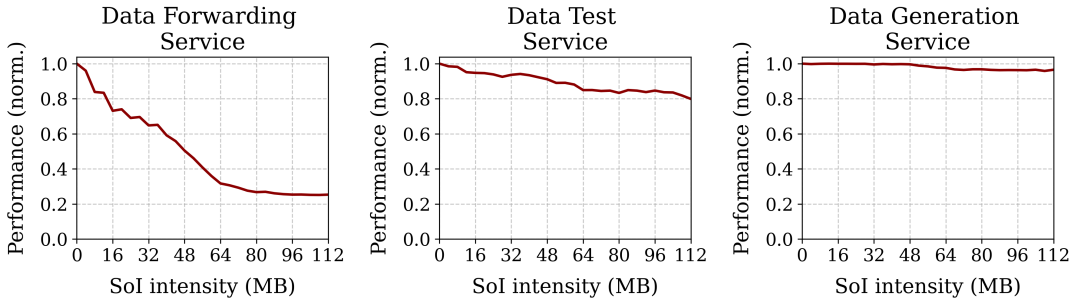


Figure 8.8: Sensitivity curves of MDS microservices

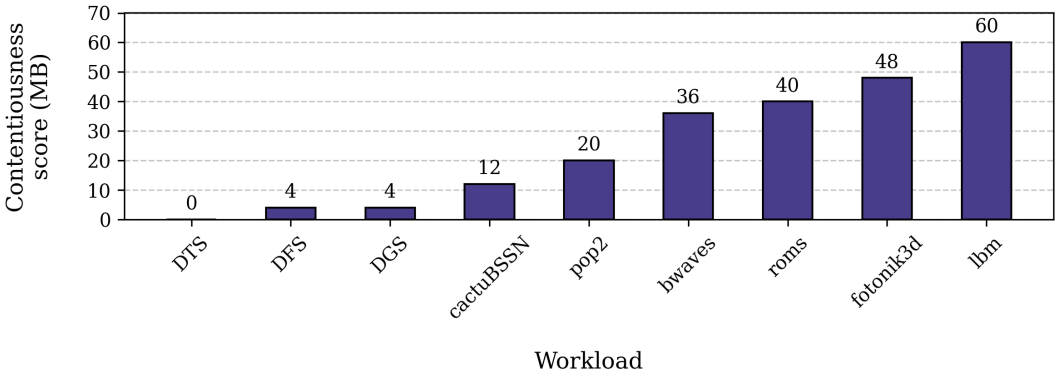


Figure 8.9: Contentiousness of MDS microservices (represented by the first three bars from the left) and SPEC benchmarks included in the MDS experimental corpus.

8.4.3 Prediction Results

To determine prediction accuracy for the MDS workloads, we compare predicted performance values with measurements obtained from an actual co-deployment. During profiling of each application, all of its dependencies are deployed on a separate Kubernetes node. This way we ensure that the application can interface with its dependencies while being isolated from mutual resource contention.

We compare the predicted and actual performance of each microservice in various co-locations in Figures 8.10 through 8.12.

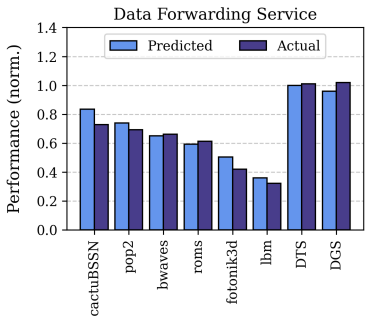


Figure 8.10: DFS validation

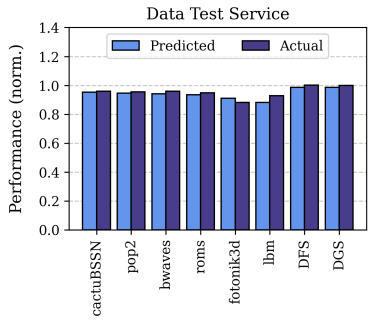


Figure 8.11: DTS validation

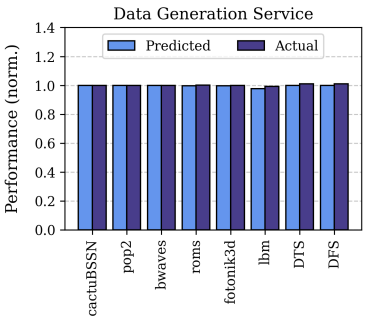


Figure 8.12: DGS validation

In each of those figures, the x-axis shows a competitor (either a MDS application or a SPEC benchmark), while the y-axis shows the performance of the microservice. The first bar repre-

sents the predicted performance, while the second the actual performance. The more similar the height of two bars is, the better the prediction. Because performance was normalized against execution in isolation, run-to-run variability can produce values greater than 1.0. As expected, the least sensitive DGS exhibits the smallest performance degradation, while the most sensitive DFS shows significant slowdown when co-located with progressively more competitive workloads.

In Figure 8.13, we illustrate the error distribution of performance prediction applied to the MDS system. The x-axis shows prediction error, that is actual performance subtracted from the expected performance. The y-axis shows the count of the values. The median absolute error is 1.4%, and the 90th percentile of the absolute error is 7.4%. The minimum and maximum error is -6% and 10.7%, respectively. Overall, the method's accuracy for predicting the performance of MDS is slightly worse than in the first case study involving only the SPEC benchmarks. We believe this may result from multiple causes, including, but not limited to: variations in the input message rate due to network instability, additional interference introduced by concurrently running Kubernetes workloads, and greater performance variability in the execution of the MDS microservices.

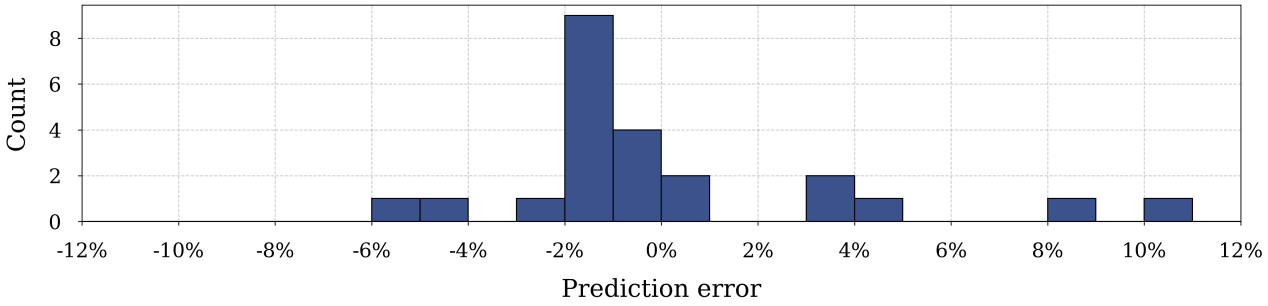


Figure 8.13: Histogram of error distribution in the MDS case study experiment

Chapter 9

Conclusion

In this chapter we summarize our unique contributions presented throughout this thesis, distill key findings derived from the discussion in Chapter 8, and draw final conclusions. First, we provide a summary of our work and answer the research questions in Section 9.1. Next, we determine threats to validity and possible ways to resolve them in Section 9.2. Finally, we recommend future work directions to fully address performance prediction in optimization of microservice-based CPS deployments in Section 9.3.

9.1 Summary

Throughout this thesis, we extend an existing methodology for predicting memory interference between co-located application pairs by proposing three unique contributions. We describe our first contribution, introducing empirically validated design guidelines for contention scoring mechanisms, in Chapter 5, and in the process answer RQ1:

> **RQ1:** *"How to design a mechanism for accurately characterizing the contentiousness of applications?"*
> **A:** "We develop and evaluate four different contentiousness reporting mechanisms, each generating a different kind of memory traffic: Type 1 uses only streaming memory accesses; Type 2 generates exclusively random memory traffic; Type 3 alternates between streaming and random accesses; and Type 4 employs a hybrid access pattern. Our experiments suggests that the quality of contentiousness characterization, quantified by the overall accuracy of the method achieved when employing a specific scoring mechanism, depends on the employed memory access pattern. Specifically, Types 1 and 2 perform poorly, with 90th-percentile absolute error that is 2.0% and 5.5% worse than that of Type 4. Type 4 provides higher accuracy than Type 3 in at least 95% of cases, while Type 3 yields the most accurate predictions at the 99th percentile. A successful contentiousness reporter should also utilize a microbenchmarking solution, which allows for accurately measuring performance degradation, and thus quantifying the resource pressure that is felt by the mechanism. We find the Benchmark[a] library to perform well for this task. Prior work suggests that the size of the memory region accessed by the mechanism should extend the capacity of the L3 cache in order to posses holistic sensitivity to memory contention. In our experiments, a working set size equal to eight times the L3 capacity yields robust predictions."

[a] `https://github.com/google/benchmark`

Next, we describe the second mentioned contribution, providing concrete recommendations for

practitioners regarding improving prediction accuracy by configuring the execution environment, in Chapter 6, and formulate the following answer for RQ2:

> **RQ2:** *"How to configure an execution environment that allows for ac curate prediction of performance degradation caused by memory subsystem contention?"*
> **A:** "Our results indicate that the configuration of CPU frequency-scaling policies substantially affects prediction accuracy. We evaluated six `CPUFreq` governors available on Linux systems: `performance`, `powersave`, `userspace`, `ondemand`, `conservative`, and `schedutil`. We also tested the effect of enabling Turbo Boost in combination with the `performance` governor. In total, we examined seven configurations and found that the `performance` governor with Turbo Boost disabled produced the most accurate predictions, with 90th-percentile accuracy 3% higher than the next-best configuration (`userspace` governor) and 15% higher than the worst-performing configuration (`performance` governor and Turbo Boost), while incurring minimal impact on application performance. Additionally, we reviewed prior work that describes techniques for isolating workloads by setting CPU affinity, which eliminates unpredictable CPU interference between co-located tasks."

We also present a third contribution, which is an open source implementation of a framework for predicting performance degradation. We discuss our implementation in Chapter 7, and release it under open source license[1].

In Chapter 8, we demonstrate the method's strong predictive performance on two case studies: industrial benchmarks (median error = 1%, 90th percentile error = 5.5%); and microservices (median error = 1.4%, 90th percentile error = 7.4%).

9.2 Threats to Validity

Lack of cross-architectural validation Our experiments were carried out on a single hardware and OS configuration. Resource contention arises from the interaction of many factors, including, but not limited to, virtualization techniques, memory subsystem structure, and specific OS behaviors, and thus may differ across platforms. Although we expect the proposed method to generalize to many multi-core architectures, its accuracy may degrade and the benefits of following our guidelines may diminish on other platforms. Therefore, evaluating the approach on a broader set of architectures is necessary to establish the wider generalizability of our claims.

Microservice corpus Since the microservices encompassed in the MDS corpus are not contentious, they do not impact each other when parts of the systems are co-located together on different nodes. If services did contend, however, their inter-dependencies could complicate the application of our method. For example, degraded performance in an upstream service may reduce the request rate to a downstream dependency and thus (paradoxically) improve the downstream service's observed performance. We think it would be valuable to re-asses our method using a larger corpus of inter-dependent microservices that also exhibit a wider spectrum of contentiousness.

9.3 Future Work

Scaling beyond pair-wise co-locations A clear shortcoming of our method is its inability to support predictions for more than two co-running applications, which restricts its applica-

[1] `https://github.com/Bruno-Dzk/resource-contention`

bility to large-scale systems. Our review of related work uncovers a potential solution using the scalability table approach proposed by Govindan et al. [8] (see Section 3.2.1).

Addition of other shared resources Another limitation of our method is that it can only estimate contention in one shared resource – the memory subsystem. Extending the variety of supported resources with CPU, network, and/or disk contention would broaden the applicability of the approach to workloads that intensely utilize these resources. This need is especially highlighted by the MDS system experiment, which involves microservices for which network is a significant bottleneck. We predict that expanding the method to additional resources will require new, resource specific SoIs and reporters. While Delimitrou and Kozyrakis [7] describe SoIs for 15 potential resources and Zhao et al. [19] address multi-threaded, CPU-intensive SoIs, none of the reviewed work provides information on reporters specialized for resources other than memory. We also expect that generalizing the method in terms of resources may require inventing a novel approach for combining resource-specific sensitivity and contentiousness models into a single, general model.

Generalizing to heterogeneous architectures As described in Section 1.2, we assume that analyzed workloads are deployed only on homogeneous nodes. Extending our method to accommodate heterogeneous architectures, represents a compelling avenue for future work. Notably, this problem has been examined in the context of interference minimization in an existing study by Delimitrou and Kozyrakis [16].

Acknowledgements

I want to acknowledge the entire TechFlex team at TNO-ESI for hosting my thesis and helping me during this incredible journey. In particular, I am very grateful to my academic supervisor, Benny Åkesson, for giving a significant amount of his time to make sure I succeeded while teaching me a great deal about the scientific process, and for always believing in me. I also want to thank Ben Pronk, my daily supervisor, for his relentless enthusiasm, technical expertise, and for coming up with excellent ideas no one else thought of. Additionally, I am very grateful to David Worell for making sure I was "happy and paid" during my internship, and to Sven Weiss for his time spent helping me with technical matters, and especially his prowess in debugging Kubernetes clusters.

My special thanks go to my fiancée, Gosia Janeczko. She listened to every worry of mine, validated my ideas, made brilliant suggestions, and helped me to see a silver lining whenever I was stuck.

I also want to thank Joseph Hill from the Informatics Institute at the University of Amsterdam for giving me access to an experimental setup that proved indispensable for my research.

Bibliography

[1] S. Acur and T. Hendriks, "Vision and outlook for systems architecting and systems engineering in the high-tech equipment industry," TNO-ESI, Tech. Rep., 2024.

[2] R. N. Charette, *How software is eating the car*, https://spectrum.ieee.org/software-eating-car, Accessed: 2025-11-13.

[3] *Software at asml*, https://www.asml.com/en/technology/software, Accessed: 2025-11-13.

[4] A. van der Werf, G. Upis, I. Oehlmann, J. W. Velthuijsen, J. Castillo, and R. Ribas Sant, "Made in nl: The value of the dutch high-tech manufacturing industry," PricewaterhouseCoopers B.V., Tech. Rep., 2024.

[5] M. Volaard, "Hardware dimensioning for microservice-based cyber-physical systems: A profiling and performance prediction method," M.S. thesis, Vrije Universiteit Amsterdam and Universiteit van Amsterdam, 2023.

[6] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011, pp. 248–259.

[7] C. Delimitrou and C. Kozyrakis, "Ibench: Quantifying interference for datacenter applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013, pp. 23–33. DOI: 10.1109/IISWC.2013.6704667.

[8] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines," *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011*, Oct. 2011. DOI: 10.1145/2038916.2038938.

[9] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, Tel-Aviv, Israel: Association for Computing Machinery, 2013, pp. 607–618, ISBN: 9781450320795. DOI: 10.1145/2485922.2485974. [Online]. Available: https://doi.org/10.1145/2485922.2485974.

[10] A. Manousis, R. A. Sharma, V. Sekar, and J. Sherry, "Contention-aware performance prediction for virtualized network functions," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 270–282, ISBN: 9781450379557. DOI: 10.1145/3387514.3405868. [Online]. Available: https://doi.org/10.1145/3387514.3405868.

[11] M. Adeppady, C. F. Chiasserini, H. Karl, and P. Giaccone, "Iplace: An interference-aware clustering algorithm for microservice placement," in *ICC 2022 - IEEE International Conference on Communications*, 2022, pp. 5457–5462. DOI: 10.1109/ICC45855.2022.9839222.

[12] M. Adeppady, P. Giaccone, H. Karl, and C. F. Chiasserini, "Reducing microservices interference and deployment time in resource-constrained cloud systems," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3135–3147, 2023. DOI: 10.1109/TNSM.2023.3235710.

[13] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," en, *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 1995.

[14] S. Mittal, "A Survey of Recent Prefetching Techniques for Processor Caches," en, *ACM Computing Surveys*, vol. 49, no. 2, pp. 1–35, Jun. 2017, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/2907071. Accessed: Nov. 5, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2907071.

[15] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M.-L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *ISCA*, 2011.

[16] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," *SIGPLAN Not.*, vol. 48, no. 4, pp. 77–88, Mar. 2013, ISSN: 0362-1340. DOI: 10.1145/2499368.2451125. [Online]. Available: https://doi.org/10.1145/2499368.2451125.

[17] W. Lin, C. Xiong, W. Wu, F. Shi, K. Li, and M. Xu, "Performance interference of virtual machines: A survey," *ACM Comput. Surv.*, vol. 55, no. 12, Mar. 2023, ISSN: 0360-0300. DOI: 10.1145/3573009. [Online]. Available: https://doi.org/10.1145/3573009.

[18] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, Nov. 2012, Conference Name: 2012 SC - International Conference for High Performance Computing, Networking, Storage and Analysis ISBN: 9781467308052 9781467308069 Place: Salt Lake City, UT Publisher: IEEE. DOI: 10.1109/SC.2012.78. Accessed: Jan. 26, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/6468475/.

[19] Y. Zhao, J. Rao, and Q. Yi, "Characterizing and Optimizing the Performance of Multithreaded Programs Under Interference," en, in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, Haifa Israel: ACM, Sep. 2016, pp. 287–297, ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967939. Accessed: Nov. 7, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2967938.2967939.

[20] R. Hood et al., "Performance impact of resource contention in multicore systems," *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–12, Apr. 2010. DOI: 10.1109/IPDPS.2010.5470399.

[21] X. Chen, L. Rupprecht, R. Osman, P. Pietzuch, F. Franciosi, and W. Knottenbelt, "Cloudscope: Diagnosing and managing performance interference in multi-tenant clouds," in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015, pp. 164–173. DOI: 10.1109/MASCOTS.2015.35.

[22] D. Huang, L. Costero, A. Pahlevan, M. Zapater, and D. Atienza, "Cloudprophet: A machine learning-based performance prediction for public clouds," *IEEE Transactions on Sustainable Computing*, vol. 9, no. 4, pp. 661–676, 2024. DOI: 10.1109/TSUSC.2024.3359325.

[23] J. Rahman and P. Lama, "Predicting the end-to-end tail latency of containerized microservices in the cloud," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, 2019, pp. 200–210. DOI: 10.1109/IC2E.2019.00034.

[24] R. Mohammad, S. Gopalakrishnan, and K. Pattabiraman, "Co-approximator: Enabling performance prediction in colocated applications.," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 1, Oct. 2024, ISSN: 1539-9087. DOI: 10.1145/3677180. [Online]. Available: https://doi.org/10.1145/3677180.

[25] R. S. Kannan, M. Laurenzano, J. Ahn, J. Mars, and L. Tang, "Caliper: Interference estimator for multi-tenant environments sharing architectural resources," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 3, Jun. 2019, ISSN: 1544-3566. DOI: 10.1145/3323090. [Online]. Available: https://doi.org/10.1145/3323090.

[26] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on hpc systems with node sharing," in *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12. DOI: 10.1145/2503210.2503256.

[27] S.-G. Kim, H. Eom, and H. Y. Yeom, "Virtual machine consolidation based on interference modeling," *J. Supercomput.*, vol. 66, no. 3, pp. 1489–1506, Dec. 2013, ISSN: 0920-8542. DOI: 10.1007/s11227-013-0939-2. [Online]. Available: https://doi.org/10.1007/s11227-013-0939-2.

[28] J. Mars and M. L. Soffa, "Synthesizing contention," en, in *Proceedings of the Workshop on Binary Instrumentation and Applications*, New York New York USA: ACM, Dec. 2009, pp. 17–25, ISBN: 978-1-60558-793-6. DOI: 10.1145/1791194.1791197. Accessed: Aug. 12, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/1791194.1791197.

[29] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming Performance Variability," en, 2018, pp. 409–425, ISBN: 978-1-939133-08-3. Accessed: Jul. 4, 2025. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/maricq.

[30] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 1–10. DOI: 10.1109/IISWC.2016.7581261. Accessed: Jul. 3, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7581261.

Appendix A

Implementation of Reporters

In this appendix, we present the implementations of the remaining three reporter types used in our study (alternating access reporter is discussed in Section 5.5). First, we present the implementation of the streaming access reporter in Listing A.1. Next, we showcase the random access reporter in Listing A.2. We conclude with the implementation of the hybrid reporter in Listing A.3.

```
#include <benchmark/benchmark.h>

#define STREAM_SIZE 16000000
#define PADDING_SIZE 400000

static double bw_data [STREAM_SIZE + 2 * PADDING_SIZE];
static double scalar = 3.0;

void streaming_access(benchmark::State& state) {
    for (auto _ : state) {
        double *mid = bw_data + PADDING_SIZE;
        for (int i = 0; i < STREAM_SIZE / 2; i++) {
            bw_data[i]= scalar*mid[i];
        }
        for (int i = 0; i< STREAM_SIZE / 2; i++) {
            mid[i]= scalar*bw_data[i];
        }
    }
}

BENCHMARK(streaming_access);
BENCHMARK_MAIN();
```

Listing A.1: Implementation of the streaming access reporter (Type 1)

```
#include <benchmark/benchmark.h>

#define FOOTPRINT_SIZE 16000000

unsigned lfsr;
#define MASK 0xd0000001u
#define rand (lfsr = (lfsr >> 1) ^ (-(int)(lfsr & 1u) & MASK))
#define r (rand % FOOTPRINT_SIZE)

static unsigned int data_chunk[FOOTPRINT_SIZE];
static double scalar = 3.0;
static unsigned int dump[10];

void random_access(benchmark::State& state) {
    for (auto _ : state) {
        lfsr = 0xACE1u;
        for (int i = 0; i < FOOTPRINT_SIZE / 100; i++) {
            dump[0] += data_chunk[r]++;
            dump[1] += data_chunk[r]++;
            dump[2] += data_chunk[r]++;
            dump[3] += data_chunk[r]++;
            dump[4] += data_chunk[r]++;
            dump[5] += data_chunk[r]++;
            dump[6] += data_chunk[r]++;
            dump[7] += data_chunk[r]++;
            dump[8] += data_chunk[r]++;
            dump[9] += data_chunk[r]++;
            benchmark::ClobberMemory();
        }
    }
}

BENCHMARK(random_access);
BENCHMARK_MAIN();
```

Listing A.2: Implementation of the random access reporter (Type 2)

```
#include <benchmark/benchmark.h>

#ifndef LBM_SIZE
  #define LBM_SIZE 16000000
#endif

typedef double LBMGrid[LBM_SIZE];
static double * srcGrid, * dstGrid;

void hybrid_access(benchmark::State& state) {
    const unsigned long margin = 400000;
    const unsigned long size = sizeof(LBMGrid) + 2 * margin * sizeof(
       double);
    srcGrid = new double[size];
    dstGrid = new double[size];
    srcGrid += margin;
    dstGrid += margin;
    for (auto _ : state) {
        for (int i = 0; i < LBM_SIZE; i += 20) {
            dstGrid[i] = srcGrid[i];
            dstGrid[i - 1998] = srcGrid[(1) + i];
            dstGrid[i + 2001] = srcGrid[(2) + i];
            dstGrid[i - 16] = srcGrid[(3) + i];
            dstGrid[i + 23] = srcGrid[(4) + i];
            dstGrid[i - 199994] = srcGrid[(5) + i];
            dstGrid[i + 200005] = srcGrid[(6) + i];
            dstGrid[i - 2010] = srcGrid[(7) + i];
            dstGrid[i - 1971] = srcGrid[(8) + i];
            dstGrid[i + 1988] = srcGrid[(9) + i];
            dstGrid[i + 2027] = srcGrid[(10) + i];
            dstGrid[i - 201986] = srcGrid[(11) + i];
            dstGrid[i + 198013] = srcGrid[(12) + i];
            dstGrid[i - 197988] = srcGrid[(13) + i];
            dstGrid[i + 202011] = srcGrid[(14) + i];
            dstGrid[i - 200002] = srcGrid[(15) + i];
            dstGrid[i + 199997] = srcGrid[(16) + i];
            dstGrid[i - 199964] = srcGrid[(17) + i];
            dstGrid[i + 200035] = srcGrid[(18) + i];
            }
        }
}

BENCHMARK(hybrid_access);
BENCHMARK_MAIN();
```

Listing A.3: Implementation of the hybrid access reporter (Type 4)

Appendix B

Results of Reporter Validation

Figures B.1–B.4 illustrate the prediction validation results for the four reporter types considered in this thesis. Each large table summarizes validation across 171 co-locations of SPEC CPU benchmarks. Applications under analysis are shown in the table columns and competitors in the rows; the cell at the intersection of application A (column) and competitor C (row) reports the relative prediction error $\varepsilon P_A(C)$, computed according to Equation 7.1 and rounded to one decimal place. Cell background color encodes the magnitude and direction of the error: increasingly intense red denotes larger overestimation of performance, while increasingly intense blue denotes larger underestimation. Beneath each table we report the maximum and minimum error values and selected percentiles of the distribution of absolute errors.

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		1.2%	2.8%	0.6%	0.2%	1.4%	0.6%	-0.3%	3.8%	0.4%	0.9%	2.4%	0.2%	-0.3%	4.9%	0.2%	1.2%	2.1%	0.7%
mcf	2.7%		6.8%	0.4%	1.2%	2.7%	1.6%	0.6%	8.6%	0.8%	2.1%	6.3%	2.3%	2.1%	-2.9%	0.5%	4.3%	4.3%	2.6%
omnetpp	2.7%	-3.3%		1.5%	1.6%	3.6%	3.0%	0.8%	7.7%	1.4%	1.3%	2.7%	2.7%	2.3%	-1.4%	1.4%	0.8%	0.3%	3.0%
xalancbmk	0.8%	1.9%	1.2%		0.1%	0.4%	0.1%	0.5%	-0.1%	0.0%	-0.4%	0.0%	-1.1%	1.0%	5.9%	0.2%	0.5%	0.1%	0.8%
x264	0.8%	-3.0%	-0.6%	-0.4%		1.3%	0.5%	-0.1%	2.4%	0.3%	0.4%	1.3%	-0.1%	0.1%	-3.9%	0.1%	0.9%	0.6%	1.1%
deepsjeng	1.0%	0.4%	2.2%	-0.2%	0.1%		0.7%	0.0%	2.0%	0.3%	3.5%	0.6%	-2.7%	-0.3%	1.4%	-0.1%	1.1%	1.5%	0.0%
leela	0.2%	0.8%	-1.3%	0.0%	0.1%	0.3%		1.9%	0.6%	0.4%	0.2%	1.1%	-0.4%	-0.2%	-1.2%	0.2%	0.2%	0.7%	0.2%
exchange2	0.2%	4.1%	0.2%	0.6%	0.2%	0.2%	0.0%		0.8%	0.0%	0.2%	0.4%	-0.3%	-0.1%	1.5%	0.0%	0.1%	0.0%	0.0%
xz	1.5%	5.6%	3.9%	-0.5%	0.5%	2.3%	1.1%	0.4%		-0.1%	1.3%	3.2%	-0.9%	1.0%	5.4%	0.3%	1.5%	1.9%	1.0%
bwaves	-11.8%	-13.8%	-10.4%	-6.7%	-5.8%	-2.6%	-4.9%	-2.8%	-8.6%		-4.5%	-1.3%	-7.1%	0.7%	-3.0%	-0.6%	-1.2%	-13.9%	-11.6%
cactuBSSN	0.7%	3.9%	1.4%	2.9%	1.8%	3.2%	2.5%	1.6%	5.9%	1.8%		6.0%	3.0%	3.7%	4.9%	0.8%	1.7%	-0.2%	3.0%
lbm	-15.3%	-5.4%	-9.6%	-15.6%	-12.3%	-3.0%	-6.7%	-3.6%	-5.3%	-2.2%	-12.8%		-7.6%	-7.3%	-15.7%	-2.5%	-5.9%	-13.0%	-19.8%
cam4	1.9%	1.9%	4.0%	1.0%	0.6%	2.4%	1.1%	0.4%	5.8%	0.7%	1.2%	6.8%		-0.2%	1.1%	0.4%	3.1%	3.5%	1.3%
pop2	0.3%	2.9%	1.5%	2.8%	1.2%	3.1%	2.5%	1.1%	3.0%	1.3%	-0.1%	7.1%	0.3%		0.1%	1.0%	1.9%	-0.6%	1.7%
imagick	1.2%	5.0%	1.9%	-0.8%	0.1%	0.9%	0.4%	-0.1%	2.6%	0.5%	0.9%	2.6%	-3.1%	2.1%		0.2%	1.0%	3.1%	0.6%
nab	0.7%	-1.2%	1.9%	0.5%	0.0%	1.2%	0.1%	1.1%	1.2%	-0.2%	0.4%	1.9%	1.3%	-0.1%	-6.9%		0.6%	0.9%	0.5%
fotonik3d	-7.9%	-8.0%	-8.8%	-6.9%	-6.0%	-1.6%	-4.3%	-2.4%	-6.9%	-2.3%	-13.2%	-13.3%	-12.8%	-6.7%	-3.5%	-0.6%		-21.4%	-11.1%
roms	-4.7%	-5.2%	-3.5%	-1.4%	-1.5%	1.0%	-0.1%	-0.9%	0.3%	0.3%	-4.7%	-5.6%	1.2%	2.9%	-3.2%	0.6%	-2.5%		-3.9%
perlbench	0.3%	0.5%	1.9%	1.1%	-0.1%	0.6%	0.5%	0.1%	1.3%	0.5%	0.3%	0.9%	2.3%	-0.1%	-1.5%	0.1%	0.7%	0.9%	

Max error	8.6%	
Min error	-21.4%	
Absolute error percentiles	0.5	1.16%
	0.9	6.32%
	0.95	8.78%
	0.99	15.66%

Figure B.1: Validation of the streaming access reporter (Type 1)

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		-5.9%	-2.5%	0.2%	-0.2%	0.6%	0.6%	-0.3%	0.4%	0.9%	-0.6%	-1.3%	-0.8%	0.4%	4.9%	0.7%	-1.0%	-2.2%	0.0%
mcf	1.1%		1.5%	0.0%	0.7%	1.9%	1.6%	0.6%	5.2%	1.3%	0.6%	2.6%	1.3%	2.8%	-2.8%	1.1%	2.0%	0.0%	1.8%
omnetpp	-0.1%	-0.1%		2.5%	1.6%	2.9%	2.6%	0.7%	6.2%	1.0%	0.3%	0.2%	4.3%	2.0%	-0.9%	1.1%	-0.8%	-2.5%	2.5%
xalancbmk	0.8%	1.9%	1.2%		0.1%	0.4%	0.1%	0.5%	-0.1%	0.0%	-0.4%	0.0%	-1.1%	1.0%	5.9%	0.2%	0.5%	0.1%	0.8%
x264	0.8%	-3.0%	-0.6%	-0.4%		1.3%	0.5%	-0.1%	2.4%	0.3%	0.4%	1.3%	-0.1%	0.1%	-3.9%	0.1%	0.9%	0.6%	1.1%
deepsjeng	1.0%	0.4%	2.2%	-0.2%	0.1%		0.7%	0.0%	2.0%	0.3%	3.5%	0.6%	-2.7%	-0.3%	1.4%	-0.1%	1.1%	1.5%	0.0%
leela	0.2%	0.8%	-1.3%	0.0%	0.1%	0.3%		1.9%	0.6%	0.4%	0.2%	1.1%	-0.4%	-0.2%	-1.2%	0.2%	0.2%	0.7%	0.2%
exchange2	0.2%	4.1%	0.2%	0.6%	0.2%	0.2%	0.0%		0.8%	0.0%	0.2%	0.4%	-0.3%	-0.1%	1.5%	0.0%	0.1%	0.0%	0.0%
xz	1.5%	5.6%	3.9%	-0.5%	0.5%	2.3%	1.1%	0.4%		-0.1%	1.3%	3.2%	-0.9%	1.0%	5.4%	0.3%	1.5%	1.9%	1.0%
bwaves	-14.2%	-15.5%	-13.8%	-9.6%	-8.4%	-4.9%	-8.9%	-3.2%	-12.3%		-8.3%	-5.6%	-8.3%	-5.2%	-4.3%	-1.5%	-5.7%	-18.8%	-13.7%
cactuBSSN	-3.7%	1.2%	-2.3%	-0.2%	-0.9%	0.5%	0.2%	0.7%	-2.5%	0.8%		1.2%	-6.0%	1.5%	-2.4%	0.1%	-1.0%	-4.4%	-1.1%
lbm	-15.3%	-5.4%	-9.6%	-15.6%	-12.3%	-3.0%	-6.7%	-3.6%	-5.3%	-2.2%	-12.8%		-7.6%	-7.3%	-15.7%	-2.5%	-5.9%	-13.0%	-19.8%
cam4	0.3%	-5.2%	-1.3%	0.6%	0.2%	1.6%	1.1%	0.4%	2.5%	1.2%	-0.2%	3.0%		0.5%	1.2%	0.9%	0.8%	-0.8%	0.5%
pop2	-7.8%	-2.3%	-9.8%	-3.8%	-3.4%	-1.8%	-2.8%	-1.7%	-6.1%	-0.6%	-6.1%	-5.4%	-3.9%		-2.9%	-0.1%	-4.5%	-10.6%	-5.9%
imagick	-0.4%	-2.1%	-3.4%	-1.2%	-0.3%	0.1%	0.4%	-0.1%	-0.8%	1.0%	-0.5%	-1.1%	-4.1%	2.8%		0.7%	-1.2%	-1.2%	-0.2%
nab	0.7%	-1.2%	1.9%	0.5%	0.0%	1.2%	0.1%	1.1%	1.2%	-0.2%	0.4%	1.9%	1.3%	-0.1%	-6.9%		0.6%	0.9%	0.5%
fotonik3d	-12.3%	-10.3%	-14.6%	-9.0%	-8.6%	-4.9%	-7.1%	-4.4%	-10.7%	-5.2%	-16.5%	-16.1%	-15.8%	-11.9%	-8.5%	-1.1%		-24.6%	-13.3%
roms	-7.1%	-6.9%	-6.9%	-4.3%	-4.1%	-1.3%	-4.1%	-1.3%	-3.4%	-0.9%	-8.5%	-9.9%	0.0%	-3.0%	-4.6%	-0.3%	-6.9%		-5.9%
perlbench	0.3%	0.5%	1.9%	1.1%	-0.1%	0.6%	0.5%	0.1%	1.3%	0.5%	0.3%	0.9%	2.3%	-0.1%	-1.5%	0.1%	0.7%	0.9%	

Max error	6.2%
Min error	-24.6%

Absolute error percentiles	
0.5	1.09%
0.9	7.76%
0.95	12.27%
0.99	17.44%

Figure B.2: Validation of the random access reporter (Type 2)

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		1.2%	2.8%	0.6%	0.2%	1.4%	0.6%	-0.3%	3.8%	0.4%	0.9%	2.4%	0.2%	-0.3%	4.9%	0.2%	1.2%	2.1%	0.7%
mcf	-1.7%		-3.4%	1.0%	0.7%	1.3%	1.3%	0.5%	3.7%	0.9%	-0.3%	0.1%	3.0%	2.6%	-2.3%	0.8%	0.4%	-2.7%	1.3%
omnetpp	2.7%	-3.3%		1.5%	1.6%	3.6%	3.0%	0.8%	7.7%	1.4%	1.3%	2.7%	2.7%	2.3%	-1.4%	1.4%	0.8%	0.3%	3.0%
xalancbmk	0.8%	1.9%	1.2%		0.1%	0.4%	0.1%	0.5%	-0.1%	0.0%	-0.4%	0.0%	-1.1%	1.0%	5.9%	0.2%	0.5%	0.1%	0.8%
x264	0.8%	-3.0%	-0.6%	-0.4%		1.3%	0.5%	-0.1%	2.4%	0.3%	0.4%	1.3%	-0.1%	0.1%	-3.9%	0.1%	0.9%	0.6%	1.1%
deepsjeng	1.0%	0.4%	2.2%	-0.2%	0.1%		0.7%	0.0%	2.0%	0.3%	3.5%	0.6%	-2.7%	-0.3%	1.4%	-0.1%	1.1%	1.5%	0.0%
leela	0.2%	0.8%	-1.3%	0.0%	0.1%	0.3%		1.9%	0.6%	0.4%	0.2%	1.1%	-0.4%	-0.2%	-1.2%	0.2%	0.2%	0.7%	0.2%
exchange2	0.2%	4.1%	0.2%	0.6%	0.2%	0.2%	0.0%		0.8%	0.0%	0.2%	0.4%	-0.3%	-0.1%	1.5%	0.0%	0.1%	0.0%	0.0%
xz	-0.1%	-1.5%	-1.4%	-0.9%	0.1%	1.5%	1.1%	0.4%		0.4%	-0.2%	-0.5%	-1.9%	1.8%	5.4%	0.9%	-0.7%	-2.4%	0.3%
bwaves	-8.0%	-8.2%	-7.0%	-5.9%	-4.2%	-0.8%	-4.0%	-1.9%	-5.6%		-2.4%	4.2%	-5.1%	0.8%	-1.9%	-0.4%	1.7%	-7.4%	-7.2%
cactuBSSN	-1.0%	2.9%	-0.5%	1.9%	1.4%	2.4%	2.3%	1.5%	2.2%	1.3%		3.7%	0.6%	2.2%	5.6%	0.9%	0.5%	-1.8%	2.4%
lbm	-1.7%	5.4%	5.8%	-6.6%	-5.2%	9.4%	0.3%	-0.7%	8.2%	9.7%	-2.3%		2.1%	5.2%	-10.4%	0.1%	5.5%	-4.6%	-9.9%
cam4	0.3%	-5.2%	-1.3%	0.6%	0.2%	1.6%	1.1%	0.4%	2.5%	1.2%	-0.2%	3.0%		0.5%	1.2%	0.9%	0.8%	-0.8%	0.5%
pop2	-3.8%	-1.8%	-1.7%	0.5%	-1.0%	0.8%	-0.3%	-0.1%	-1.7%	0.8%	-2.5%	3.2%	-6.5%		1.0%	0.5%	0.1%	-4.0%	-1.3%
imagick	-0.4%	-2.1%	-3.4%	-1.2%	-0.3%	0.1%	0.4%	-0.1%	-0.8%	1.0%	-0.5%	-1.1%	-4.1%	2.8%		0.7%	-1.2%	-1.2%	-0.2%
nab	0.7%	-1.2%	1.9%	0.5%	0.0%	1.2%	0.1%	1.1%	1.2%	-0.2%	0.4%	1.9%	1.3%	-0.1%	-6.9%		0.6%	0.9%	0.5%
fotonik3d	7.7%	0.1%	13.6%	3.2%	5.5%	7.9%	4.3%	0.9%	8.3%	6.9%	0.5%	16.2%	-2.6%	9.3%	3.9%	1.6%		6.0%	5.2%
roms	-7.1%	-6.9%	-6.9%	-4.3%	-4.1%	-1.3%	-4.1%	-1.3%	-3.4%	-0.9%	-8.5%	-9.9%	0.0%	-3.0%	-4.6%	-0.3%	-6.9%		-5.9%
perlbench	0.3%	0.5%	1.9%	1.1%	-0.1%	0.6%	0.5%	0.1%	1.3%	0.5%	0.3%	0.9%	2.3%	-0.1%	-1.5%	0.1%	0.7%	0.9%	

Max error	16.2%
Min error	-10.4%

Absolute error percentiles	
0.5	0.98%
0.9	5.45%
0.95	7.17%
0.99	10.44%

Figure B.3: Validation of the alternating access reporter (Type 3)

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		1.2%	2.8%	0.6%	0.2%	1.4%	0.6%	-0.3%	3.8%	0.4%	0.9%	2.4%	0.2%	-0.3%	4.9%	0.2%	1.2%	2.1%	0.7%
mcf	2.7%		6.8%	0.4%	1.2%	2.7%	1.6%	0.6%	8.6%	0.8%	2.1%	6.3%	2.3%	2.1%	-2.9%	0.5%	4.3%	4.3%	2.6%
omnetpp	2.7%	-3.3%		1.5%	1.6%	3.6%	3.0%	0.8%	7.7%	1.4%	1.3%	2.7%	2.7%	2.3%	-1.4%	1.4%	0.8%	0.3%	3.0%
xalancbmk	0.8%	1.9%	1.2%		0.1%	0.4%	0.1%	0.5%	-0.1%	0.0%	-0.4%	0.0%	-1.1%	1.0%	5.9%	0.2%	0.5%	0.1%	0.8%
x264	0.8%	-3.0%	-0.6%	-0.4%		1.3%	0.5%	-0.1%	2.4%	0.3%	0.4%	1.3%	-0.1%	0.1%	-3.9%	0.1%	0.9%	0.6%	1.1%
deepsjeng	1.0%	0.4%	2.2%	-0.2%	0.1%		0.7%	0.0%	2.0%	0.3%	3.5%	0.6%	-2.7%	-0.3%	1.4%	-0.1%	1.1%	1.5%	0.0%
leela	0.2%	0.8%	-1.3%	0.0%	0.1%	0.3%		1.9%	0.6%	0.4%	0.2%	1.1%	-0.4%	-0.2%	-1.2%	0.2%	0.2%	0.7%	0.2%
exchange2	0.2%	4.1%	0.2%	0.6%	0.2%	0.2%	0.0%		0.8%	0.0%	0.2%	0.4%	-0.3%	-0.1%	1.5%	0.0%	0.1%	0.0%	0.0%
xz	1.5%	5.6%	3.9%	-0.5%	0.5%	2.3%	1.1%	0.4%		-0.1%	1.3%	3.2%	-0.9%	1.0%	5.4%	0.3%	1.5%	1.9%	1.0%
bwaves	-8.0%	-8.2%	-7.0%	-5.9%	-4.2%	-0.8%	-4.0%	-1.9%	-5.6%		-2.4%	4.2%	-5.1%	0.8%	-1.9%	-0.4%	1.7%	-7.4%	-7.2%
cactuBSSN	-0.5%	2.2%	-0.6%	1.2%	0.3%	1.7%	1.3%	1.5%	0.6%	1.0%		2.3%	0.7%	2.7%	5.0%	0.4%	-0.5%	-3.3%	0.5%
lbm	-6.1%	3.0%	-0.1%	-8.7%	-7.7%	6.1%	-2.5%	-2.6%	4.4%	6.8%	-5.6%		-0.9%	0.1%	-15.4%	-0.4%	1.1%	-7.9%	-12.2%
cam4	0.3%	-5.2%	-1.3%	0.6%	0.2%	1.6%	1.1%	0.4%	2.5%	1.2%	-0.2%	3.0%		0.5%	1.2%	0.9%	0.8%	-0.8%	0.5%
pop2	0.3%	2.9%	1.5%	2.8%	1.2%	3.1%	2.5%	1.1%	3.0%	1.3%	-0.1%	7.1%	0.3%		0.1%	1.0%	1.9%	-0.6%	1.7%
imagick	1.2%	5.0%	1.9%	-0.8%	0.1%	0.9%	0.4%	-0.1%	2.6%	0.5%	0.9%	2.6%	-3.1%	2.1%		0.2%	1.0%	3.1%	0.6%
nab	0.7%	-1.2%	1.9%	0.5%	0.0%	1.2%	0.1%	1.1%	1.2%	-0.2%	0.4%	1.9%	1.3%	-0.1%	-6.9%		0.6%	0.9%	0.5%
fotonik3d	-3.9%	-6.7%	-1.7%	-4.2%	-2.3%	2.0%	-1.6%	-1.3%	-2.0%	0.5%	-9.5%	-7.1%	-10.5%	-2.4%	0.4%	0.0%		-15.6%	-5.1%
roms	-4.7%	-5.2%	-3.5%	-1.4%	-1.5%	1.0%	-0.1%	-0.9%	0.3%	0.3%	-4.7%	-5.6%	1.2%	2.9%	-3.2%	0.6%	-2.5%		-3.9%
perlbench	0.3%	0.5%	1.9%	1.1%	-0.1%	0.6%	0.5%	0.1%	1.3%	0.5%	0.3%	0.9%	2.3%	-0.1%	-1.5%	0.1%	0.7%	0.9%	

Max error	8.6%	
Min error	-15.6%	
Absolute error percentiles	0.5	1.01%
	0.9	4.96%
	0.95	6.75%
	0.99	11.18%

Figure B.4: Validation of the hybrid access reporter (Type 4)

Appendix C

Results of CPU Frequency Scaling Experiments

Figures C.1–C.6 illustrate the prediction validation results for the six `CPUFreq` governors considered in this thesis, while Figure 9.7 represents experimental results obtained with the Turbo Boost mechanism coupled with the `performance` governor. Each large table summarizes validation across 171 co-locations of SPEC CPU benchmarks. Applications under analysis are shown in the table columns and competitors in the rows; the cell at the intersection of application A (column) and competitor C (row) reports the relative prediction error $\varepsilon P_A(C)$, computed according to Equation 7.1 and rounded to one decimal place. Cell background color encodes the magnitude and direction of the error: increasingly intense red denotes larger overestimation of performance, while increasingly intense blue denotes larger underestimation. Beneath each table we report the maximum and minimum error values and selected percentiles of the distribution of absolute errors.

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		1.2%	2.8%	0.6%	0.2%	1.4%	0.6%	-0.3%	3.8%	0.4%	0.9%	2.4%	0.2%	-0.3%	4.9%	0.2%	1.2%	2.1%	0.7%
mcf	-1.7%		-3.4%	1.0%	0.7%	1.3%	1.3%	0.5%	3.7%	0.9%	-0.3%	0.1%	3.0%	2.6%	-2.3%	0.8%	0.4%	-2.7%	1.3%
omnetpp	2.7%	-3.3%		1.5%	1.6%	3.6%	3.0%	0.8%	7.7%	1.4%	1.3%	2.7%	2.7%	2.3%	-1.4%	1.4%	0.8%	0.3%	3.0%
xalancbmk	0.8%	1.9%	1.2%		0.1%	0.4%	0.1%	0.5%	-0.1%	0.0%	-0.4%	0.0%	-1.1%	1.0%	5.9%	0.2%	0.5%	0.1%	0.8%
x264	0.8%	-3.0%	-0.6%	-0.4%		1.3%	0.5%	-0.1%	2.4%	0.3%	0.4%	1.3%	-0.1%	0.1%	-3.9%	0.1%	0.9%	0.6%	1.1%
deepsjeng	1.0%	0.4%	2.2%	-0.2%	0.1%		0.7%	0.0%	2.0%	0.3%	3.5%	0.6%	-2.7%	-0.3%	1.4%	-0.1%	1.1%	1.5%	0.0%
leela	0.2%	0.8%	-1.3%	0.0%	0.1%	0.3%		1.9%	0.6%	0.4%	0.2%	1.1%	-0.4%	-0.2%	-1.2%	0.2%	0.2%	0.7%	0.2%
exchange2	0.2%	4.1%	0.2%	0.6%	0.2%	0.2%	0.0%		0.8%	0.0%	0.2%	0.4%	-0.3%	-0.1%	1.5%	0.0%	0.1%	0.0%	0.0%
xz	-0.1%	-1.5%	-1.4%	-0.9%	0.1%	1.5%	1.1%	0.4%		0.4%	-0.2%	-0.5%	-1.9%	1.8%	5.4%	0.9%	-0.7%	-2.4%	0.3%
bwaves	-8.0%	-8.2%	-7.0%	-5.9%	-4.2%	-0.8%	-4.0%	-1.9%	-5.6%		-2.4%	4.2%	-5.1%	0.8%	-1.9%	-0.4%	1.7%	-7.4%	-7.2%
cactuBSSN	-1.0%	2.9%	-0.5%	1.9%	1.4%	2.4%	2.3%	1.5%	2.2%	1.3%		3.7%	0.6%	2.2%	5.6%	0.9%	0.5%	-1.8%	2.4%
lbm	-1.7%	5.4%	5.8%	-6.6%	-5.2%	9.4%	0.3%	-0.7%	8.2%	9.7%	-2.3%		2.1%	5.2%	-10.4%	0.1%	5.5%	-4.6%	-9.9%
cam4	0.3%	-5.2%	-1.3%	0.6%	0.2%	1.6%	1.1%	0.4%	2.5%	1.2%	-0.2%	3.0%		0.5%	1.2%	0.9%	0.8%	-0.8%	0.5%
pop2	-3.8%	-1.8%	-1.7%	0.5%	-1.0%	0.8%	-0.3%	-0.1%	-1.7%	0.8%	-2.5%	3.2%	-6.5%		1.0%	0.5%	0.1%	-4.0%	-1.3%
imagick	-0.4%	-2.1%	-3.4%	-1.2%	-0.3%	0.1%	0.4%	-0.1%	-0.8%	1.0%	-0.5%	-1.1%	-4.1%	2.8%		0.7%	-1.2%	-1.2%	-0.2%
nab	0.7%	-1.2%	1.9%	0.5%	0.0%	1.2%	0.1%	1.1%	1.2%	-0.2%	0.4%	1.9%	1.3%	-0.1%	-6.9%		0.6%	0.9%	0.5%
fotonik3d	7.7%	0.1%	13.6%	3.2%	5.5%	7.9%	4.3%	0.9%	8.3%	6.9%	0.5%	16.2%	-2.6%	9.3%	3.9%	1.6%		6.0%	5.2%
roms	-7.1%	-6.9%	-6.9%	-4.3%	-4.1%	-1.3%	-4.1%	-1.3%	-3.4%	-0.9%	-8.5%	-9.9%	0.0%	-3.0%	-4.6%	-0.3%	-6.9%		-5.9%
perlbench	0.3%	0.5%	1.9%	1.1%	-0.1%	0.6%	0.5%	0.1%	1.3%	0.5%	0.3%	0.9%	2.3%	-0.1%	-1.5%	0.1%	0.7%	0.9%	

Max error	16.2%	
Min error	-10.4%	
Absolute error percentiles	0.5	0.98%
	0.9	5.45%
	0.95	7.17%
	0.99	10.44%

Figure C.1: Validation with the `performance` governor

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		0.5%	-0.6%	0.2%	-0.3%	0.4%	0.4%	-0.3%	1.3%	1.1%	-1.4%	0.8%	2.5%	0.6%	-0.8%	0.1%	-1.3%	-2.3%	-0.2%
mcf	0.6%		2.1%	1.5%	0.4%	1.6%	1.2%	0.8%	5.5%	1.8%	0.3%	3.4%	-0.3%	4.2%	-13.4%	0.3%	2.0%	0.2%	2.2%
omnetpp	1.7%	1.2%		2.1%	2.0%	3.0%	2.8%	1.0%	8.4%	2.3%	-0.3%	3.8%	3.6%	4.9%	-7.7%	0.6%	0.9%	0.3%	3.8%
xalancbmk	-2.4%	0.2%	-2.7%		-0.5%	-0.7%	-0.2%	-0.5%	-1.4%	0.5%	-1.6%	-2.3%	-1.0%	4.6%	1.8%	-0.1%	-2.1%	-3.6%	-0.2%
x264	-1.5%	0.4%	-2.2%	0.5%		0.0%	0.1%	0.2%	0.3%	1.0%	-1.9%	-1.6%	0.7%	1.6%	-3.8%	0.3%	-2.1%	-2.8%	0.4%
deepsjeng	-1.2%	0.7%	-2.6%	0.8%	0.0%		0.4%	-0.4%	-0.4%	0.7%	0.7%	-2.1%	-0.6%	2.6%	-4.8%	0.2%	-2.3%	-2.8%	0.0%
leela	-1.2%	-4.2%	-4.6%	0.1%	-0.4%	-0.8%		-0.7%	-1.9%	0.7%	-1.9%	-2.3%	1.8%	4.0%	-6.7%	-0.2%	-2.4%	-3.5%	-0.4%
exchange2	-1.6%	-0.5%	-4.4%	0.4%	-0.2%	-0.8%	0.0%		-2.2%	0.4%	-2.1%	-3.0%	0.3%	3.3%	0.0%	-0.1%	-2.7%	-3.8%	-0.6%
xz	-0.9%	1.3%	-1.4%	0.5%	-0.2%	0.9%	0.5%	1.0%		0.8%	-1.1%	0.4%	1.4%	1.9%	-6.6%	0.3%	-0.7%	-1.7%	0.6%
bwaves	-12.0%	-0.7%	-9.4%	-5.7%	-6.8%	-3.2%	-6.4%	-1.6%	-7.5%		-6.2%	-1.2%	-3.6%	-2.7%	-3.0%	-1.6%	0.0%	-14.1%	-10.9%
cactuBSSN	0.1%	-2.2%	0.4%	1.9%	0.7%	2.4%	1.8%	0.8%	3.2%	1.0%		4.3%	-1.4%	0.2%	0.6%	-0.1%	0.8%	-2.6%	0.2%
lbm	-11.9%	-4.9%	-6.6%	-13.6%	-12.6%	-0.8%	-8.1%	-4.6%	-4.2%	-2.8%	-13.8%		-4.7%	-8.4%	-13.3%	-2.9%	-5.3%	-11.1%	-14.4%
cam4	0.1%	1.6%	0.3%	0.6%	0.4%	1.2%	0.8%	0.0%	2.4%	1.3%	-0.5%	3.0%		3.6%	-7.4%	0.4%	0.1%	-1.2%	1.1%
pop2	-4.2%	-0.3%	-4.3%	0.3%	-1.3%	-0.2%	0.2%	-0.2%	-1.6%	0.6%	-0.1%	5.3%	-5.9%		-0.7%	0.0%	0.9%	-3.4%	-2.9%
imagick	-1.8%	-1.4%	-2.4%	0.2%	-0.2%	0.0%	0.1%	-0.6%	0.9%	1.1%	-0.8%	-0.9%	-3.1%	2.7%		0.2%	-1.4%	-1.1%	-0.1%
nab	-0.9%	-3.8%	-2.1%	-0.3%	-0.1%	0.2%	0.1%	0.0%	0.6%	0.6%	-1.1%	-2.0%	-1.9%	1.3%	-3.7%		-1.9%	-3.6%	-0.1%
fotonik3d	-9.4%	-7.9%	-9.9%	-8.7%	-6.1%	-3.7%	-5.9%	-3.6%	-5.6%	-1.7%	-13.1%	-14.7%	-12.0%	-10.9%	-10.9%	-1.5%		-21.4%	-9.5%
roms	-7.1%	-4.5%	-8.3%	-4.2%	-4.6%	-1.6%	-2.8%	-1.2%	-4.2%	-1.5%	-8.0%	-12.7%	-1.6%	-4.5%	-4.8%	-0.8%	-6.6%		-6.2%
perlbench	-1.7%	-0.1%	-3.6%	0.4%	0.0%	-0.3%	-0.1%	-0.4%	-0.6%	0.7%	-1.6%	-1.8%	0.7%	3.2%	-5.5%	-0.1%	-2.0%	-3.0%	

Max error	8.4%
Min error	-21.4%
Absolute error percentiles	0.5 1.30%
	0.9 6.64%
	0.95 9.51%
	0.99 14.25%

Figure C.2: Validation with the `userspace` governor

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		-5.6%	-1.9%	-0.3%	-0.5%	0.3%	0.0%	0.1%	0.8%	2.8%	1.3%	-0.5%	-6.7%	3.2%	2.0%	0.6%	-0.1%	-0.7%	0.5%
mcf	2.0%		0.2%	-0.4%	1.3%	1.5%	0.9%	0.5%	5.3%	2.6%	1.1%	3.2%	0.0%	3.3%	0.7%	0.4%	3.3%	0.7%	1.3%
omnetpp	3.0%	-3.2%		1.9%	1.5%	3.0%	2.6%	1.7%	7.0%	2.9%	2.6%	2.9%	2.0%	1.7%	0.7%	0.7%	2.1%	1.4%	3.6%
xalancbmk	-1.8%	-9.9%	-3.1%		-0.3%	-0.3%	-0.5%	-0.1%	-2.2%	2.0%	-0.7%	-3.1%	-3.9%	1.5%	8.0%	0.1%	-1.5%	-2.5%	-0.9%
x264	-0.4%	-1.6%	-3.4%	0.5%		-0.2%	0.0%	0.5%	-0.8%	1.5%	-0.6%	-2.0%	-6.0%	1.7%	7.3%	0.1%	-1.2%	-1.5%	-0.5%
deepsjeng	-0.6%	-5.0%	-2.8%	0.5%	-0.1%		-0.1%	0.4%	-0.4%	2.3%	4.0%	-2.6%	-9.2%	0.9%	-4.0%	0.6%	-1.4%	-1.6%	-0.9%
leela	-0.6%	-6.0%	-5.6%	-0.1%	-1.0%	-0.8%		-0.6%	-2.0%	0.7%	0.2%	-2.3%	-0.4%	1.1%	-1.5%	-0.1%	-1.4%	-2.7%	-0.2%
exchange2	-1.2%	-10.3%	-4.0%	0.3%	-0.2%	-1.0%	-0.5%		-2.6%	1.9%	-0.4%	-3.2%	-3.4%	1.9%	0.1%	-0.1%	-2.1%	-3.0%	-0.9%
xz	-0.4%	-7.3%	-1.2%	0.3%	0.2%	0.8%	0.1%	0.9%		2.7%	0.5%	-0.4%	-3.7%	1.8%	4.4%	0.5%	0.7%	-0.4%	0.7%
bwaves	-8.8%	-11.5%	-4.6%	-5.1%	-4.8%	-2.4%	-4.8%	-1.6%	-6.2%		-2.5%	4.3%	1.6%	0.5%	-0.2%	-1.1%	3.4%	-7.1%	-11.0%
cactuBSSN	1.9%	3.5%	0.2%	2.2%	1.1%	3.1%	2.2%	0.9%	5.2%	1.4%		4.8%	-0.6%	3.2%	-1.2%	0.1%	2.5%	1.5%	2.7%
lbm	-12.7%	-5.4%	-5.3%	-13.6%	-11.4%	-1.9%	-8.6%	-4.6%	-5.2%	-2.4%	-8.9%		-11.2%	-7.3%	-12.6%	-2.2%	-4.5%	-10.5%	-15.9%
cam4	1.0%	-4.4%	-0.1%	0.1%	0.3%	1.1%	0.5%	0.5%	2.4%	1.9%	3.2%	2.4%		2.1%	0.2%	0.4%	2.1%	0.5%	0.7%
pop2	-4.0%	-7.7%	-4.5%	-0.9%	-0.8%	0.1%	-0.7%	-0.6%	-2.5%	0.3%	0.2%	2.9%	-6.4%		0.6%	-0.1%	0.7%	-2.4%	-3.1%
imagick	-0.6%	-4.9%	-1.9%	0.3%	-0.4%	-0.2%	-0.1%	-0.1%	-0.7%	2.1%	1.9%	-1.3%	-6.5%	1.4%		0.1%	-0.3%	0.2%	-0.1%
nab	-0.9%	-5.3%	-2.8%	0.5%	-0.2%	0.3%	-0.1%	0.1%	0.0%	0.9%	-0.2%	-2.0%	-3.8%	-0.6%	-1.1%		-1.2%	-3.2%	-0.4%
fotonik3d	-9.5%	-8.1%	-8.0%	-7.1%	-5.4%	-2.9%	-4.7%	-4.0%	-6.3%	-2.2%	-9.8%	-15.7%	-21.1%	-9.1%	-8.4%	-1.1%		-19.4%	-11.5%
roms	-7.6%	-8.9%	-8.2%	-5.0%	-4.7%	-3.1%	-3.2%	-3.3%	-4.0%	-1.4%	-3.2%	-12.6%	-0.4%	-4.7%	-4.2%	-1.2%	-8.3%		-7.9%
perlbench	-1.4%	-10.0%	-3.7%	-0.3%	-1.1%	-0.5%	-0.1%	0.3%	-1.4%	1.7%	-0.7%	-2.6%	-2.3%	1.0%	2.3%	0.2%	-1.5%	-1.8%	

Max error	8.0%
Min error	-21.1%
Absolute error percentiles	0.5 1.47%
	0.9 7.29%
	0.95 9.55%
	0.99 15.76%

Figure C.3: Validation with the `ondemand` governor

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		-3.8%	0.7%	0.4%	0.2%	0.7%	0.5%	0.3%	0.0%	1.2%	3.9%	-0.4%	5.2%	0.8%	-5.2%	0.5%	-1.1%	0.0%	0.5%
mcf	0.9%		1.8%	1.1%	1.3%	2.1%	1.4%	-0.5%	5.3%	3.1%	4.9%	2.7%	4.3%	1.7%	-5.2%	1.1%	2.3%	1.9%	2.2%
omnetpp	2.3%	-5.3%		1.6%	2.0%	3.6%	3.0%	1.9%	8.0%	2.0%	2.0%	3.0%	10.5%	4.4%	-6.0%	1.6%	0.9%	1.4%	3.7%
xalancbmk	-1.9%	-4.3%	-4.2%		-0.1%	-0.4%	-0.1%	-0.3%	-3.8%	1.3%	-0.1%	-2.9%	6.9%	1.0%	-4.6%	0.6%	-2.1%	-0.9%	-1.1%
x264	-1.2%	-8.1%	-2.9%	0.2%		0.6%	0.3%	0.2%	-1.1%	1.6%	1.6%	-2.4%	8.6%	1.7%	-11.4%	0.5%	-1.6%	-1.1%	-0.2%
deepsjeng	-1.0%	-1.5%	-2.6%	0.5%	0.7%		0.1%	0.2%	-1.3%	1.7%	3.6%	-3.5%	9.2%	2.4%	-13.1%	0.8%	-1.7%	-1.0%	-0.4%
leela	-2.2%	-6.4%	-3.3%	-0.5%	-0.2%	-0.3%		-0.1%	-2.6%	1.1%	0.6%	-2.6%	9.2%	0.4%	-8.3%	0.8%	-2.1%	-1.9%	-0.9%
exchange2	-1.7%	-4.5%	-4.4%	-3.5%	0.0%	-0.7%	0.0%		-3.7%	1.2%	2.0%	-3.9%	5.9%	1.2%	-7.9%	0.6%	-2.2%	-1.6%	0.1%
xz	-0.3%	-5.2%	-1.6%	0.5%	0.3%	1.4%	0.7%	0.2%		1.9%	2.1%	-0.5%	12.7%	2.5%	-0.6%	0.7%	-0.7%	0.6%	1.1%
bwaves	-8.2%	-6.6%	-5.0%	-5.6%	-4.6%	-1.1%	-3.5%	-1.6%	-6.8%		-2.0%	4.4%	8.9%	-0.5%	-14.4%	-0.5%	1.1%	-8.3%	-6.8%
cactuBSSN	-0.2%	-7.2%	0.3%	2.7%	1.0%	2.3%	1.7%	0.6%	2.5%	1.2%		3.4%	13.4%	0.0%	-9.1%	0.9%	0.4%	-2.8%	2.0%
lbm	-17.5%	-12.9%	-9.1%	-14.7%	-12.5%	-5.3%	-9.2%	-4.2%	-5.8%	-1.6%	-9.5%		2.3%	-8.2%	-11.2%	-2.2%	-5.5%	-12.0%	-15.6%
cam4	-0.3%	-5.6%	-0.7%	0.5%	0.9%	1.5%	1.1%	0.4%	1.2%	2.1%	4.5%	2.8%		2.1%	-3.9%	1.0%	0.3%	0.9%	1.4%
pop2	-1.7%	-0.3%	-3.3%	-0.1%	0.1%	1.3%	1.4%	0.6%	0.2%	2.6%	3.5%	5.7%	4.6%		-5.5%	0.7%	0.7%	-2.3%	-0.7%
imagick	-0.5%	-6.0%	-0.3%	-0.7%	0.1%	0.2%	0.3%	-0.1%	-0.8%	1.8%	2.0%	-1.1%	7.1%	1.6%		0.7%	-1.6%	0.8%	0.0%
nab	-1.3%	-10.1%	-2.5%	0.7%	0.3%	0.5%	0.0%	0.0%	-0.6%	1.4%	0.9%	-0.5%	5.6%	0.1%	-5.8%		-1.6%	-0.8%	0.9%
fotonik3d	-8.4%	-6.5%	-6.3%	-6.8%	-4.9%	-3.4%	-6.4%	-3.4%	-7.9%	-1.4%	-12.3%	-13.3%	-5.3%	-6.6%	-13.9%	-0.7%		-20.5%	-9.8%
roms	-4.5%	-7.1%	-3.3%	-2.3%	-2.7%	-0.3%	-1.8%	0.2%	-1.2%	2.3%	-3.8%	-3.7%	4.6%	-0.9%	-8.0%	0.2%	-2.8%		-2.7%
perlbench	-1.8%	-10.7%	-4.4%	-0.4%	0.1%	-0.3%	0.2%	-0.2%	-1.3%	2.0%	0.8%	-2.2%	9.4%	1.9%	-0.4%	0.8%	-1.9%	-0.5%	

Max error	13.4%	
Min error	-20.5%	
Absolute error percentiles	0.5	1.59%
	0.9	7.96%
	0.95	10.49%
	0.99	15.02%

12.05%	10.47%		
-31.05%	-20.45%	-10.60%	

Figure C.4: Validation with the **`schedutil`** governor

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		1.3%	-0.5%	-0.5%	-1.2%	-0.1%	0.2%	-0.1%	0.7%	0.6%	-0.2%	0.3%	-5.8%	0.2%	-0.7%	0.1%	-0.2%	-3.4%	-0.6%
mcf	0.5%		1.1%	1.4%	0.8%	1.3%	1.0%	0.9%	19.9%	1.2%	0.8%	3.4%	-5.9%	4.8%	-4.8%	0.4%	3.3%	-1.2%	2.1%
omnetpp	1.9%	-3.3%		1.3%	0.8%	2.8%	2.3%	1.8%	8.9%	0.8%	0.2%	3.3%	2.2%	4.3%	-0.4%	0.3%	0.9%	-1.1%	4.7%
xalancbmk	-1.4%	-7.4%	-4.7%		-1.1%	6.1%	-0.2%	0.3%	-1.2%	1.1%	-1.5%	-2.3%	-11.7%	0.5%	0.1%	-0.4%	-1.3%	-4.9%	-2.3%
x264	-2.0%	1.5%	-3.4%	-0.6%		-0.5%	-0.2%	0.6%	0.5%	0.0%	-1.5%	-1.4%	-0.3%	-0.5%	0.0%	-0.5%	-1.5%	-4.4%	-1.6%
deepsjeng	-1.0%	-1.8%	-1.9%	-0.6%	-0.4%		0.0%	0.5%	-0.1%	0.0%	2.7%	-2.4%	-17.4%	0.0%	2.9%	-0.5%	-1.3%	-4.6%	-1.8%
leela	-2.2%	-1.7%	-3.0%	-0.4%	-1.0%	-0.8%		0.2%	-2.0%	1.1%	-2.6%	-2.3%	-9.9%	1.3%	-0.2%	-0.5%	-1.5%	-5.2%	-0.3%
exchange2	-3.1%	-6.9%	-5.4%	-0.2%	-0.4%	-1.5%	-0.3%		-1.8%	0.1%	-1.6%	-2.8%	-17.7%	0.6%	0.2%	-0.3%	-1.6%	-5.3%	-1.3%
xz	-0.8%	-1.8%	-2.2%	0.4%	-0.3%	0.5%	0.6%	0.7%		0.9%	0.2%	0.1%	-9.7%	-0.1%	-4.8%	-0.3%	0.3%	-3.5%	-0.2%
bwaves	-10.9%	-5.9%	-9.9%	-7.0%	-6.2%	-3.0%	-6.1%	-2.1%	-7.7%		-6.1%	-0.1%	-11.4%	-5.6%	-1.7%	-1.9%	0.0%	-14.3%	-10.7%
cactuBSSN	0.4%	-3.2%	-0.3%	1.6%	0.0%	1.8%	1.7%	0.6%	2.7%	0.8%		4.4%	-7.0%	-0.2%	2.0%	0.6%	1.7%	-2.6%	1.4%
lbm	-13.9%	-9.9%	-10.2%	-14.5%	-15.3%	-5.5%	-11.2%	-4.9%	-8.5%	-6.1%	-11.9%		-14.2%	-8.6%	-10.6%	-4.0%	-7.1%	-14.0%	-17.8%
cam4	-0.5%	-1.4%	0.4%	-0.2%	-0.3%	0.5%	0.5%	0.6%	2.9%	0.5%	-0.6%	3.1%		0.9%	-5.4%	-0.1%	0.7%	-2.3%	0.6%
pop2	-4.9%	-7.8%	-5.9%	-1.8%	-2.5%	-1.1%	-2.5%	-1.1%	-5.0%	-1.1%	-1.6%	0.0%	3.3%		-5.0%	-0.2%	-1.6%	-8.4%	-2.9%
imagick	-0.5%	-5.6%	-2.1%	0.7%	-0.5%	-0.5%	0.4%	-0.2%	0.7%	-0.4%	1.3%	-0.9%	-4.3%	-0.2%		-0.4%	-1.2%	-2.4%	-1.5%
nab	-1.1%	-6.4%	-2.7%	0.7%	-0.4%	-0.2%	0.5%	0.0%	1.0%	-0.1%	0.6%	-1.5%	-13.7%	0.1%	-2.9%		-1.3%	-5.4%	0.1%
fotonik3d	-9.1%	-6.5%	-8.7%	-9.1%	-6.5%	-1.5%	-4.2%	-2.2%	-6.0%	-3.7%	-11.9%	-15.1%	-10.9%	-9.4%	-13.2%	-1.2%		-23.9%	-10.8%
roms	-17.0%	-6.2%	-6.2%	-3.4%	-4.1%	-2.0%	-2.3%	-1.6%	-3.5%	-2.0%	-6.6%	-12.5%	-7.9%	-8.0%	-8.2%	-1.0%	-8.3%		-7.5%
perlbench	-2.4%	-6.8%	-3.0%	-0.1%	-0.5%	-1.0%	-0.3%	-0.6%	-0.8%	1.1%	-0.8%	-1.5%	-4.6%	1.5%	-0.1%	-0.1%	-1.6%	-4.9%	

Max error	19.9%	
Min error	-23.9%	
Absolute error percentiles	0.5	1.44%
	0.9	8.60%
	0.95	11.67%
	0.99	17.73%

Figure C.5: Validation with the **`conservative`** governor

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		-11.0%	-9.3%	-12.4%	-10.4%	-10.1%	-11.1%	-9.3%	-11.3%	-7.1%	-12.5%	-9.7%	-12.2%	-9.6%	-13.9%	-11.2%	-7.5%	-6.8%	-10.9%
mcf	-9.0%		-5.0%	-11.0%	-8.8%	-8.8%	-10.2%	-8.1%	-8.0%	-7.6%	-12.8%	-10.0%	-11.0%	-7.3%	-14.1%	-10.3%	-7.0%	-4.6%	-9.7%
omnetpp	-2.6%	0.0%		-4.3%	-1.9%	-0.7%	-2.8%	0.5%	0.6%	0.3%	-6.3%	-3.2%	-1.4%	-2.8%	-4.8%	-2.8%	-2.0%	-2.3%	-0.5%
xalancbmk	-12.1%	-7.7%	-10.1%		-10.8%	-10.4%	-11.6%	-9.0%	-13.5%	-6.9%	-14.1%	-15.5%	-10.7%	-11.6%	-12.1%	-10.9%	-8.4%	-7.6%	-10.5%
x264	-11.2%	-5.2%	-8.8%	-13.2%		-10.3%	-11.4%	-8.9%	-12.7%	-7.3%	-13.3%	-14.6%	-9.1%	-11.7%	-12.0%	-11.0%	-7.9%	-7.8%	-10.9%
deepsjeng	-11.0%	-8.3%	-7.7%	-11.1%	-10.8%		-11.1%	-8.4%	-11.6%	-6.9%	-8.4%	-15.0%	-12.4%	-7.9%	-13.4%	-10.9%	-7.8%	-7.2%	-10.4%
leela	-11.5%	-11.4%	-8.8%	-12.7%	-11.2%	-10.5%		-9.1%	-13.8%	-6.8%	-12.9%	-12.9%	-10.5%	-11.2%	-11.8%	-10.9%	-8.1%	-7.6%	-10.5%
exchange2	-12.6%	-13.0%	-8.5%	-12.1%	-11.3%	-10.8%	-12.8%		-13.7%	-8.2%	-14.1%	-14.1%	-16.3%	-9.5%	-16.7%	-11.2%	-8.7%	-8.4%	-12.4%
xz	-8.2%	-5.3%	-4.4%	-9.0%	-6.2%	-5.9%	-8.2%	-4.6%		-1.5%	-9.9%	-7.6%	-9.6%	-7.8%	-12.2%	-9.0%	-6.1%	-5.4%	-5.8%
bwaves	-8.9%	-5.4%	-1.2%	-10.2%	-9.2%	-7.4%	-9.7%	-8.1%	-5.6%		-10.3%	-5.8%	-11.5%	-6.9%	-18.5%	-10.5%	-2.8%	-2.0%	-8.6%
cactuBSSN	-7.6%	-6.9%	-3.0%	-8.7%	-8.0%	-5.0%	-8.5%	-7.0%	-5.6%	-7.1%		-7.7%	-13.5%	-8.2%	-11.3%	-9.8%	-6.0%	-4.3%	-6.7%
lbm	-11.8%	-10.8%	-9.0%	-17.9%	-17.4%	-10.9%	-14.8%	-15.7%	-10.9%	-13.3%	-13.2%		-12.8%	-8.4%	-19.7%	-20.9%	-7.0%	-8.2%	-15.4%
cam4	-9.8%	-3.2%	-6.3%	-12.7%	-10.3%	-8.5%	-10.4%	-9.0%	-10.4%	-7.0%	-9.4%	-10.9%		-9.7%	-13.0%	-9.9%	-6.9%	-6.4%	-9.3%
pop2	-2.3%	0.4%	2.2%	-4.7%	-3.7%	-0.7%	-3.7%	-1.3%	-0.8%	-2.1%	-2.6%	-2.2%	-4.2%		-5.6%	-4.0%	-1.4%	-1.1%	-0.9%
imagick	-11.0%	-5.3%	-6.3%	-11.8%	-10.2%	-9.5%	-10.9%	-8.4%	-11.6%	-8.3%	-11.8%	-14.5%	-13.2%	-11.1%		-10.5%	-8.3%	-7.4%	-10.3%
nab	-11.1%	-8.2%	-6.6%	-11.7%	-10.1%	-9.8%	-10.9%	-7.6%	-13.0%	-8.8%	-11.0%	-15.2%	-12.8%	-10.6%	-13.4%		-11.1%	-9.9%	-9.6%
fotonik3d	-8.4%	-9.0%	-7.2%	-8.9%	-7.4%	-5.8%	-7.3%	-7.4%	-7.2%	-6.1%	-7.8%	-13.2%	-3.5%	-7.8%	-13.9%	-10.5%		-11.8%	-8.9%
roms	-3.5%	-0.6%	-1.0%	-4.5%	-3.9%	-2.3%	-4.7%	-4.4%	-0.9%	-3.3%	-1.7%	-7.6%	-1.9%	-2.8%	-6.8%	-7.3%	-3.3%		-3.7%
perlbench	-11.3%	-10.8%	-9.1%	-12.5%	-10.5%	-10.7%	-11.4%	-9.3%	-13.1%	-7.4%	-12.8%	-12.9%	-11.8%	-9.2%	-14.3%	-11.1%	-7.3%	-7.9%	

Max error	2.2%
Min error	-20.9%
Absolute error percentiles	0.5 8.80%
	0.9 12.97%
	0.95 14.08%
	0.99 18.17%

Figure C.6: Validation with the `powersave` governor

	gcc	mcf	omnetpp	xalancbmk	x264	deepsjeng	leela	exchange2	xz	bwaves	cactuBSSN	lbm	cam4	pop2	imagick	nab	fotonik3d	roms	perlbench
gcc		-16.5%	-5.4%	-13.9%	-13.7%	-9.4%	-13.4%	-11.9%	-6.8%	-13.3%	-18.1%	-1.8%	-3.4%	-15.3%	-15.8%	-19.2%	-3.4%	-12.9%	-11.9%
mcf	-12.2%		-2.1%	-13.1%	-13.5%	-7.7%	-12.1%	-11.4%	-4.6%	-12.0%	-17.1%	0.7%	-1.1%	-15.5%	-24.1%	-20.0%	-2.6%	-11.9%	-9.6%
omnetpp	-10.1%	-14.4%		-10.5%	-12.9%	-5.0%	-10.0%	-9.9%	1.3%	-10.7%	-15.9%	1.2%	-1.1%	-15.7%	-22.7%	-19.5%	-0.4%	-9.9%	-7.1%
xalancbmk	-15.2%	-20.2%	-9.7%		-14.6%	-11.9%	-14.2%	-11.3%	-12.6%	-14.6%	-19.0%	-5.6%	-4.2%	-17.6%	-24.7%	-18.6%	-5.8%	-16.0%	-12.4%
x264	-14.5%	-13.3%	-7.9%	-13.4%		-10.1%	-13.7%	-11.7%	-9.4%	-14.4%	-18.6%	-4.0%	1.7%	-17.9%	-17.7%	-19.2%	-5.0%	-15.1%	-11.9%
deepsjeng	-13.7%	-14.5%	-6.9%	-14.3%	-14.1%		-13.7%	-11.4%	-8.8%	-14.9%	-16.0%	-4.5%	-8.1%	-17.0%	-21.7%	-19.9%	-4.5%	-14.2%	-12.6%
leela	-15.2%	-16.1%	-10.0%	-13.9%	-14.5%	-12.1%		-10.9%	-12.8%	-14.5%	-19.0%	-5.1%	-4.1%	-16.6%	-9.0%	-19.8%	-6.4%	-16.5%	-12.9%
exchange2	-15.9%	-18.4%	-10.5%	-13.8%	-14.4%	-12.4%	-14.2%		-13.6%	-14.7%	-19.1%	-6.0%	-6.4%	-18.1%	-16.1%	-20.0%	-6.3%	-16.7%	-13.1%
xz	-13.7%	-15.0%	-5.8%	-13.7%	-13.7%	-8.8%	-12.6%	-11.3%		-13.2%	-18.0%	-1.6%	-20.9%	-13.8%	-14.6%	-18.0%	-4.2%	-13.5%	-11.6%
bwaves	-25.9%	-17.9%	-13.7%	-21.6%	-23.5%	-14.2%	-20.6%	-20.2%	-17.9%		-22.7%	-7.9%	-13.8%	-12.6%	-29.9%	-13.3%	-5.1%	-21.0%	-27.6%
cactuBSSN	-14.3%	-18.5%	-7.6%	-13.0%	-14.6%	-8.0%	-12.4%	-15.1%	-8.7%	-7.4%		1.0%	-16.3%	-9.1%	-17.8%	-11.2%	-2.2%	-11.9%	-14.3%
lbm	-21.7%	-14.6%	-7.9%	-30.1%	-30.1%	-7.7%	-19.4%	-19.4%	-9.5%	-2.8%	-20.3%		-16.6%	-9.7%	-20.4%	-11.9%	-4.5%	-12.4%	-30.6%
cam4	-11.4%	-13.6%	-2.3%	-12.6%	-12.0%	-7.1%	-11.0%	-10.1%	-3.9%	-10.4%	-15.0%	3.4%		-13.7%	-16.8%	-18.2%	0.7%	-10.6%	-9.3%
pop2	-18.8%	-19.8%	-11.4%	-16.1%	-17.3%	-9.7%	-13.5%	-17.0%	-12.6%	-5.9%	-18.7%	-2.7%	-20.7%		-11.3%	-10.3%	-2.7%	-15.9%	-19.4%
imagick	-8.1%	-14.2%	-4.0%	-7.4%	-7.0%	-6.1%	-7.2%	-7.2%	-4.8%	-5.3%	-7.3%	-1.8%	-2.8%	-5.4%		-7.1%	-3.3%	-5.6%	-5.3%
nab	-7.5%	-8.4%	-4.5%	-4.7%	-7.1%	-4.9%	-6.1%	-5.8%	-3.8%	-3.7%	-7.0%	-2.1%	-6.6%	-6.6%	-1.5%		-3.3%	-7.6%	-4.9%
fotonik3d	-21.3%	-23.4%	-9.6%	-21.2%	-21.0%	-12.2%	-21.5%	-18.2%	-10.3%	-12.1%	-28.0%	-16.7%	-18.2%	-20.3%	-31.1%	-16.6%		-25.4%	-21.6%
roms	-12.9%	-12.6%	-2.8%	-11.1%	-13.9%	-6.3%	-10.8%	-12.1%	-5.1%	-3.9%	-11.9%	-7.2%	-14.4%	-7.6%	-9.4%	-6.9%	-4.1%		-11.4%
perlbench	-14.6%	-15.7%	-9.4%	-12.6%	-14.1%	-11.3%	-13.9%	-11.7%	-11.5%	-14.1%	-18.6%	-4.2%	-5.6%	-17.5%	-12.0%	-18.4%	-5.3%	-15.2%	

Max error	3.4%
Min error	-31.1%
Absolute error percentiles	0.5 12.05%
	0.9 19.77%
	0.95 21.50%
	0.99 30.07%

Figure 9.7: Validation with Turbo Boost coupled with the `performance` governor