Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems

Dimosthenis Masouros, *Member, IEEE*,
Sotirios Xydis, *Member, IEEE*, and Dimitrios Soudris, *Member, IEEE*

Abstract—Modern micro-service and container-based cloud-native applications have leveraged multi-tenancy as a first class system design concern. The increasing number of co-located services/workloads into server facilities stresses resource availability and system capability in an unconventional and unpredictable manner. To efficiently manage resources in such dynamic environments, run-time observability and forecasting are required to capture workload sensitivities under differing interference effects, according to applied co-location scenarios. While several research efforts have emerged on interference-aware performance modelling, they are usually applied at a very coarse-grained manner e.g., estimating the overall performance degradation of an application, thus failing to effectively quantify, predict or provide educated insights on the impact of continuous runtime interference on per-resource allocations. In this paper, we present Rusty, a predictive monitoring system that leverages the power of Long Short-Term Memory networks to enable fast and accurate runtime forecasting of key performance metrics and resource stresses of cloud-native applications under interference. We evaluate Rusty under a diverse set of interference scenarios for a plethora of representative cloud workloads, showing that Rusty i) achieves extremely high prediction accuracy, average R^2 value of 0.98, ii) enables very deep prediction horizons retaining high accuracy, e.g., R^2 of around 0.99 for a horizon of 1 sec ahead and around 0.94 for an horizon of 5 sec ahead, while iii) satisfying, at the same time, the strict latency constraints required to make Rusty practical for continuous predictive monitoring at runtime.

Index Terms—predictive monitoring, system predictability, LSTM networks, interference aware, multi-tenant systems

1 Introduction

Over the last few years, the number of workloads executed on the Cloud has increased rapidly and is expected to grow more in the future [1]. The rise of cloud-native platforms, such as Kubernetes [2], that facilitate the deployment of applications on lightweight containers and expand their capacity to dynamically scale resources, further raises the density of modern cloud systems. Moreover, current Cloud solutions, such as Amazon AWS [3], Google Cloud [4], Microsoft Azure [5] and others, provide users with elasticity and resizability of their computing capacity, leading to a dynamic provisioning of resources.

This increment in the density and dynamicity of cloud workloads implies that DC operators should perform advanced resource allocation techniques, to provide both better quality-of-service (QoS) to their users, as well as maximize their profit. However, this two-factor optimization goal is rather challenging, since maximizing performance requires applications to be executed in isolation, whereas profit increment is achieved through multi-tenant job scheduling.

Recent scientific works have proposed schedulers and resource allocation techniques able to efficiently place workloads into physical/virtual servers as well as minimize their slowdown caused by multi-tenant job scheduling [6], [7], [8], [9]. Even though such approaches improve the overall performance of co-located workloads, they usually operate at a quite coarse-grained level, i.e., either requiring prior knowledge through offline characterization of isolated executions [6], [8] or in the best case gathering and aggregating runtime performance metrics through periods of enforced execution stalling/pausing to estimate interference effects [10], [11]. In addition, they fail to measure, model or exploit the dynamic impact of each specific resource on the performance degradation caused by interference, thus imposing either low accuracy estimations and/or significant performance overheads [12], [13].

To have a better understanding of the real bottlenecks of the system and specify the root cause of performance degradation, one should take a closer look at lower level architectural events and at system-level characterization to get insights regarding the state of the system [14]. Towards achieving the above goals, monitoring and analysis of system signals from within the data-center has been proven to be really beneficial and insightful. For example, Google has identified that monitoring low-level performance counters can drive better scheduling resource management and scheduling decisions [15]. In addition, the highly dynamic characteristics of cloud applications require very small monitoring and reaction times, e.g., with 1 second data sampling granularity becoming the new gold standard in cloud deployments [16].

Some hardware-enabled approaches [17], [18], [19] enabling continuous and fine-grained interference effects estimation have been proposed in the past, however their implementation requires specialization of processor's hardware, limiting their applicability in current and near future servers. Interestingly,

• *The authors are with the Department of Electrical and Computer Engineering, National Technical University of Athens, 15780 Athens, Greece. E-mail: {demo.masouros, sxydis, dsoudris}@microlab.ntua.gr.*

Manuscript received 8 Jan. 2020; revised 1 July 2020; accepted 26 July 2020. Date of publication 4 Aug. 2020; date of current version 17 Aug. 2020.
(Corresponding author: Dimosthenis Masouros.)
Recommended for acceptance by R. Ge.
Digital Object Identifier no. 10.1109/TPDS.2020.3013948

this shift towards time granularity shrinking, is also reflected in recent micro-architectural and system hardware advancements that allow fine-grained resource tuning. For example, Intel's Cache Allocation Technology in the latest Xeon processors provides software-programmable control over the amount of cache space that can be consumed by a given thread, or container [20]. Moreover, containers provide control over the exact amount of CPU and RAM resources committed to applications, while power capping frameworks [21] enable direct and fine-grained power allocation policies to be applied.

To take advantage of these low-level features, runtime monitoring that feeds and guides the scheduling algorithms should be able to dynamically predict per application resource needs in order to enforce optimised online decisions. Prior work has shown that applications experience different phases throughout their lifetime [22], [23], [24], which lead to erratic behaviors regarding memory access patterns and CPU utilization. Such behaviors become even more inconsistent when considering the interference caused by co-located applications. Therefore, fine-grained run-time predictability is crucial to elevate online resource management decisions.

From the above discussion, it is evident that new sophisticated monitoring solutions are needed to efficiently handle the emerging field of cloud-native applications. Exploiting the underlying hardware infrastructure capabilities, application and infrastructure monitoring should shift towards providing i) faster observability, to perceive the extreme diversity and dynamism in the variable workloads, and ii) continuous runtime and interference-aware predictability, i.e., to drive resource allocation decisions in a more educated manner. During the last years, several innovative monitoring solutions have raised funding in cloud industry to address the aforementioned requirements, focusing mainly on fast event logging at scale while also offering machine-learning powered analytic capabilities [16], [25], [26], [27]. Seer and Cloudseer [28], [29] form similar advanced monitoring solutions proposed from academia. However, the supported predictive analytic capabilities of those frameworks are quite coarse, imposing an "intercept-measure-and-predict" scheme to infer interference effects, thus failing to efficiently support continuous fine-grained monitoring, ideally required in modern resource allocation schemes [11], [24], [30]. Even though extended research has focused on run-time system predictability, those approaches mostly rely on the application of relatively simple empirical or regression models [31], [32], [33], being usually reactive in action, not capturing interference and neglecting the long-term dependencies and recurrence found in system level monitoring signals, thus reducing their predictive capabilities.

In this paper, we present Rusty, a monitoring framework that leverages Long Short-Term Memory (LSTM) networks to enable fast and accurate resource and energy consumption predictions of a system under interference. The novel contributions of this work are:

- We provide the first non-intermittent predictive monitoring system that is able to forecast low-level performance counters of a system under interference. Specifically, through Rusty, we are able to predict the IPC and Last-Level Cache (LLC) misses of applications running concurrently in a multi-core and, also, the energy consumption of it. Opposed to prior-art approaches that are based on short online micro-benchmarking to model interference [6], [24], [34], Rusty utilizes LSTM predictive capability to enable continuous on-the-fly predictions of interference-aware performance metrics.
- We deliver an interference-aware analysis on the low-level metrics distributions from a large pool of diverse cloud workloads, i.e., the scikit-learn [35], the Cloudsuite [36] and the SPEC2006 [37] benchmark suites. The aforementioned analysis decomposes interference effects in a per-resource manner offering an in-depth view on the sensitivity of system metrics to different stressing scenarios.
- In contrast to existing approaches, e.g., [16], [25], [28], that focus on a straightforward appliance of regression models, in this paper, we provide in-depth analysis and specific insights on LSTM parameters that affect the accuracy and complexity of the model. Through systematic exploration, analysis and fine-tuning of the LSTM's architecture and hyper-parameters, Rusty enables extremely lightweight predictive models to be deployed and operate online for continuous monitoring and adaptation.
- We evaluate Rusty in terms of its prediction precision and also demonstrate the superiority of Rusty's LSTM network over simpler machine learning approaches. Our experimental results show that Rusty exhibits very high prediction accuracy, i.e., average R^2 value of 0.98 and enables very deep prediction horizons retaining high precision, e.g., R^2 of 0.99 for a horizon of 1 sec ahead and around 0.94 for an horizon of 5 sec ahead. Finally, we show that Rusty is both i) really lightweight, introducing minimal time overheads and providing predictions in terms of milliseconds ii) and also, that it can be seamlessly transferred between systems of diverse specifications, with a slight decline in accuracy, thus forming a promising solution for runtime predictive resource allocation.

The rest of the article is organized as follows. Section 2 provides an in-depth discussion of related works. Section 3 describes our experimental setup and also demonstrates the aforementioned per-resource interference analysis. Section 4 presents our proposed predictive monitoring framework, Rusty and finally, Sections 5 and 6 show our experimental results and conclude the paper, providing future directions, respectively.

2 Related Work

Performance monitoring in data-center multi-core server architectures is essential to provide insights regarding both the load that the cluster's nodes experience and the progress of workloads being executed at the data-center's facilities.

System Monitoring. Today, state-of-the-art orchestrators, such as Kubernetes [2] and Mesos [38], rely on naive metrics to manage workloads. However, much research has been conducted regarding monitoring approaches [39] and several frameworks have been developed to enable lightweight logging and fusion of micro-architectural events [40], [41].

Moreover, services like Prometheus [42] and Amazon Cloudwatch [43] allow for custom monitoring and optimization of running workloads, forming a promising area for low-level monitoring tools. Predicting future system statuses utilizing runtime monitors has been in the center of interest of many scientific publications, with an emphasis on energy/power consumption. The authors in [31] propose a systematic methodology for assembling power models based on performance counter monitors, while in [32], [44] prediction models are proposed for effective runtime power characterisation. Although increased prediction accuracies are reported, the impact of interference is either neglected or silently suppressed, making the efficacy of the existing approaches questionable under emerging system scenarios under interference.

Performance Prediction & Control. Predicting and controlling performance degradation of workloads executing under interference has been in the center of attention of many research groups, either by scheduling workloads arriving on a cluster [6], [13], [45], [46], [47], or by regulating resource occupation by workloads throughout their execution lifetime [7], [8], [11], [24], [30], [34], [48] to improve per-application performance. A significant portion of these approaches require a priori knowledge of the target applications running on the cluster [9], [46], [47], [48], [49] or are application-specific [13], [50], thus making them unsuitable for cases of public clouds, where un-characterized applications might arrive. Other scientific approaches have built benchmark warehouses that can be used to build performance models of applications [51]. However, such static approaches do not consider the dynamic system flunctuations and thus cannot be used for continuous resource tuning. Another considerable part detects interference and performance degradation of applications by gradually pausing collocated workloads in a coarse-[8], [10], [11], [34] or fine-[24] grained manner, followed by either injecting synthetic microbenchmarks to identify resource contention or by evaluating the performance degradation compared to the isolated execution. Those approaches effectively predict resource interference, nevertheless, pausing of applications can lead to relentless efficiency decline, considering that modern servers feature up to 100 cores and, thus, can host numerous workloads simultaneously. Finally, some recent research works focus on controlling and tuning workloads in a feedback-like manner, where applications are gradually provided resources until no QoS violations are witnessed [30], [52]. Although outside the scope of the paper, Rusty can be used complementary with such control systems, as an oracle, to capture system dynamics.

Low-Level Metrics. Exploiting performance characteristics of a system through hardware performance counters has been identified as a prominent step for improving the efficiency of data centers [15]. In addition, prior works has shown that hardware performance counters can also improve the scheduling policies inside modern NUMA multi-cores [53]. Even though the importance of low-level performance counters has been pinpointed, there has been minimal work on how to "make profit" out of them, or even how these values are affected from interference effects. While state-of-the-art cloud benchmark suites [54], [55], [56] present information regarding low-level metrics of the included workloads, they provide no analysis regarding the variation of these metrics when the workloads are executed in multi-tenant environments. In addition, state-of-the-art research efforts focus mainly either on mining, extracting and reporting such low-level counters [57], or on utilizing them in a coarse-grained manner to monitor resources of virtual machines and identify anomalies or security attacks [58]. Our work, Rusty, utilizes low-level counters "on-the-fly" to provide application specific, runtime performance predictions under interference.

LSTMs in System Research. Lately, both academia and industry are shifting towards big data analytics to uncover hidden patterns, correlations and other insights behind large amount of information. In this direction, researchers are beginning to explore the employment of LSTM networks for different micro-architectural tasks. In [59], authors propose a dynamic voltage frequency scaling algorithm which uses LSTM networks to predict the workload of cores in an upcoming time period, based on performance counters collected during the previous one. Neither interference not recurrence have been taken into account and also the evaluation is based on simulated workloads and not on real system settings. In [60], the authors utilize LSTM networks in order to build efficient memory pre-fetchers and show that the employment of LSTMs for this task outperforms the state-of-the-art of traditional hardware prefetchers. Finally, in [28], the authors use a heavy-weighted LSTM and CNN network to detect QoS violations of microservices running in the cloud. Even though Seer also utilizes LSTMs, the authors apply them for debugging purposes of cloud workloads, mainly dealing with a classification problem for identifying QoS violations, thus not exploring LSTM's runtime forecasting capabilities, which are useful for predictive resource allocation. In this paper, we propose and explore the efficacy and robustness of LSTMs for fine-grained system-level predictability under resource interference, clearly extending [59] and being orthogonal to [60] and [28].

TABLE 1
Target System Specifications

Processor Model	Intel® Xeon® E5-2658A v3
Cores per socket	12 (24 logical) @2.20GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	256KB
L3 Cache	30MB, 20-way set-associative
Memory	256GB @2133MHz
Operating System	Ubuntu 16.04, kernel v4.4

3 Experimental Setup and Specifications

3.1 Target System Characterization

Rusty targets multi-core server systems usually found in DC environments. All of our experiments have been performed on a high-end server infrastructure, outlined in Table 1. In order to monitor low-level performance counters, we utilize the Performance Counter Monitoring (PCM) API [41], a tool initially developed by Intel and currently maintained in a separate github repository [61], which provides a plethora of hardware performance counters for each logical core, each socket, as well as the whole server system. We focus on specific performance counters, which we discuss in detail in Section 3.2. To simulate a cloud environment, all the referenced benchmarks running in the system have been containerized, utilizing the Docker platform [62]. Moreover, to be able to extract per-workload metrics, all the applications are assigned on a randomly selected core of the system, using the affinity rules of the docker engine.

3.2 Target Metrics Characterization

Emphasis is given to low-level system metrics, provided directly from the PCM tool [61]. For the rest of the paper, we focus on the following three performance counters, however our framework can be utilized for any metric provided by the PCM tool:

- *Instructions Per Cycle (IPC):* IPC gives insight information of the performance of the executed workload and its predictability can assist to dynamically boost/relax resources to meet certain constraints. In prior works, IPC has been used as a metric of interest to depict performance related behaviors in data center environments [10], [12], [24], [63]. For example, in [24], IPC is used to determine phase changes of applications executed under interference, whereas in [12] it used as a performance indicator of co-located workloads.
- *Last-Level Cache misses (L3M):* Memory access is considered a major bottleneck in performance. Even in modern NUMA architectures inefficient memory contention management can lead to a severe performance degradation [64]. Especially in data-center environments, it has been shown that the huge instruction sets of cloud workloads are between one and two orders of magnitude larger than the L1 instruction cache can store, and can lead to repeating instruction cache misses, which damage performance [65], [66], whereas latest reports from large-scale clouds show that memory is becoming the new bottleneck, destroying performance of applications [67]. The Last-Level Cache is basically the ”bridge” between cores requesting data and memory storing them. Last-Level Cache misses provide first-level details regarding the interference, since higher values in LLC misses can depict the tendency of multiple applications running on the system competing for access on the main memory [68].
- *Socket's energy consumption (NRG):* Energy consumption is considered as a first class constraint in modern data-center deployments [69], [70]. Accurate energy predictions at runtime significantly impact energy proportionality of the DC's nodes [71], as well as allow for the deployment of more sophisticated power capping strategies [72], [73]. In the same direction, controlling the energy and temperature levels of a system improves resiliency and components lifetimes, as well as it also reduces the cost required for cooling, which is often amounted to 50 percent of a DC's overall costs.

3.3 Workload Characterization

Modern data-center server machines accommodate a large and wide range of workloads, which are basically either batch/best-effort (BE) applications, or user-interactive/latency-critical (LC) applications. The former type of workloads require the highest possible throughput, whereas the latter demand to meet their QoS constraints. In order to cover both BE and LC workloads, we consider workloads from three popular scientific benchmarking libraries, i.e., sci-kit learn [35] and SPEC2006 [37] (as BE) and cloudsuite [36] (as LC) suites.

Through scikit-learn we examine workload skeletons that are representative of modern machine learning applications. Each instance performs the training phase of the specific workload, with datasets comprised of 40.000 instances with 784 features per instance. The spec2006 benchmarks represent computational heavy workloads as well as every-day operations performed in the cloud, e.g., *bzip2* performs compression and decompression of data entirely in memory. We use the default configurations and datasets provided by the spec2006 suite. Finally, the cloudsuite benchmarks represent services hosted in modern data-center cloud environments. The Data Serving relies on the Yahoo! Cloud Serving Benchmark [74] and the Cassandra data store [75]. In-Memory Analytics utilize Apache Spark [76] and runs a collaborative filtering algorithm on a dataset of movie ratings. Media Streaming utilizes NGINX [77] as a server for streaming videos of various lengths and qualities. Moreover, Web-Search depends on Apache Solr [78] search platform and simulates real-world clients that send requests to index nodes. Finally, Web-Serving utilizes three servers, an NGINX [77] web server, a Memcached [79] caching server and a MySQL [80] database server, simulating modern services hosted on the cloud. For the cloudsuite benchmarks, we use the default configurations and datasets as provided by the respective github repository [81] and for client-server benchmarks, we focus and monitor the server-side workload. Table 2 summarizes the benchmarks used throughout the paper. For each of the examined applications, Table 2 reports the mean value of 6 system wide low-level metrics, i.e., characterizing the IPC, on-chip cache memory behavior (**L2** cache **M**isses and **L3** cache **M**isses), energy consumption (**NRG**) and memory I/O (bytes **R**ea**D** from and bytes **WR**itten to memory), when executed in isolation, sampled every 100 milliseconds. For the rest of the paper, each workload will be identified by its ID (e.g., SK1 for AdaBoost Classifier) for simplicity.

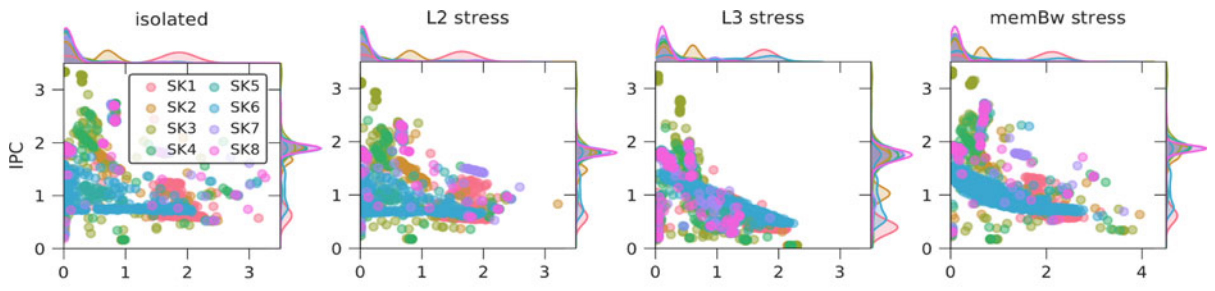
Impact of Interference on Low-Level Metrics. To showcase the sensitivity of the considered workloads w.r.t. differing resource interference, we utilize the iBench suite [82], which provides contentious micro-benchmarks, each of which stresses a different shared resource in a multi-core chip (processing cores, cache capacity and memory capacity and bandwidth). Specifically, for the following analysis, we spawn resource-specific iBench micro-benchmarks (i.e., L2 stress, L3 stress and memBw stress) to demonstrate the impact of per-resource interference on our metrics of interest.

Fig. 1 illustrates the tendency of IPC and LLC misses with differing resource stress interference, showing the high diversity in terms of IPC and L3 cache misses between our target workloads. This analysis reveals three major upshots. First, all examined workloads are insensitive to interference at the level of L2 cache, since the distributions of both IPC and LLC are slightly shifted in all cases. Second, interference at the LLC of the system induces the highest performance degradation, realizing LLC as one of the major bottlenecks in modern server systems. This is even more obvious for the cloudsuite benchmarks, where, there is a clear spread and shift of the IPC distributions towards 0, even though similar services have

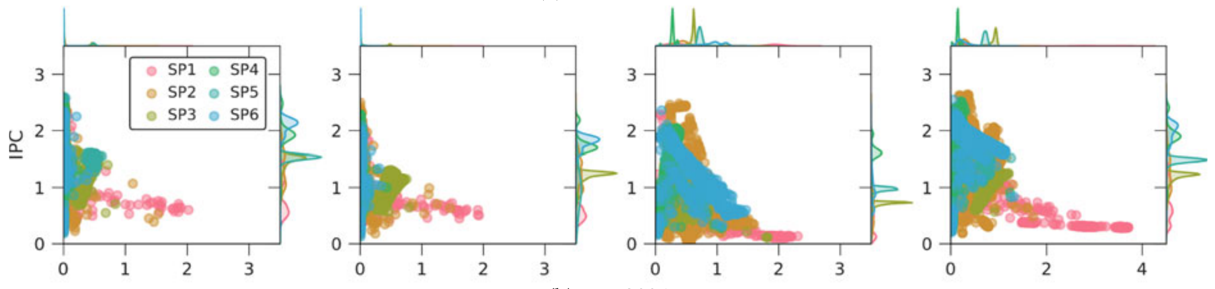
TABLE 2
Benchmarks Used as Representative Cloud Applications

	ID	Benchmark	IPC	L2M(M)	L3M(M)	NRG(J)	RD(GB)	WR(GB)
scikit-learn [35]	SK1	AdaBoost Classifier	1.10	2.42	1.21	4.21	0.10	0.01
	SK2	Lasso	1.77	0.88	0.35	4.34	0.17	0.01
	SK3	Linear Discriminant Analysis	1.94	1.17	0.18	4.28	0.05	0.03
	SK4	Linear Regression	1.83	0.91	0.16	4.23	0.04	0.02
	SK5	Random Forest Classifier	1.78	0.64	0.16	4.39	0.03	0.01
	SK6	Random Forest Regressor	1.29	2.69	0.22	4.29	0.02	0.01
	SK7	Stochastic Gradient Descent Classifier	1.85	0.46	0.21	4.32	0.06	0.01
	SK8	Stochastic Gradient Descent Regressor	1.86	0.39	0.13	4.19	0.03	0.01
spec2006 [37]	SP1	astar	0.86	0.86	0.03	4.20	0.006	0.003
	SP2	bzip2	1.34	1.24	0.02	4.17	0.006	0.004
	SP3	cactusADM	1.40	1.19	0.43	4.29	0.071	0.027
	SP4	h264ref	1.91	0.27	0.01	4.25	0.005	0.002
	SP5	leslie3d	1.50	0.85	0.47	4.32	0.322	0.153
	SP6	sphinx3	2.05	1.45	0.01	4.20	0.004	0.002
cloudsuite [36]	CS1	Data Serving	0.62	1.37	0.14	3.93	0.009	0.004
	CS2	In-Memory	1.45	0.96	0.18	4.24	0.033	0.026
	CS3	Media Streaming	0.55	0.04	0.02	2.26	0.022	0.002
	CS4	Web-Search	0.50	0.02	0.01	2.66	0.006	0.003
	CS5	Web Serving - Nginx	0.54	0.29	0.15	2.74	0.005	0.002
	CS6	Web Serving - Memcached	0.36	0.03	0.01	2.74	0.005	0.002
	CS7	Web Serving - MySQL	0.58	0.07	0.01	2.74	0.005	0.002

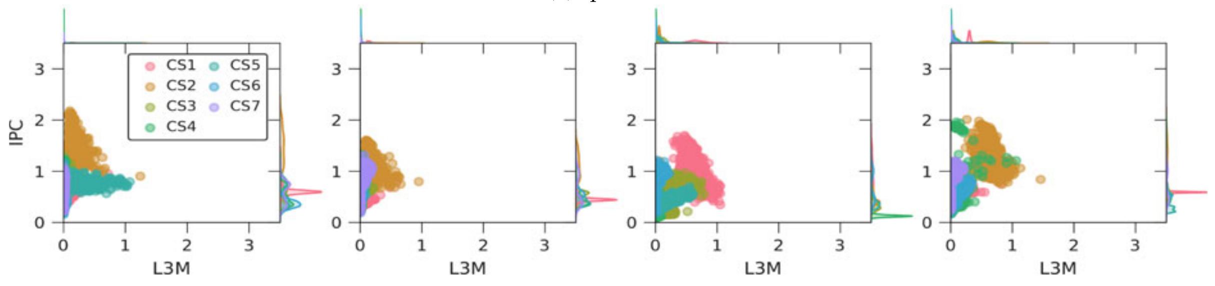
Scikit-learn and spec2006 workloads can be considered as Best-Effort (BE) applications, whereas cloudsuite workloads as Latency-Critical (LC) ones. Columns 4-9 report the average Instructions Per Cycle (IPC), L2 cache misses (L2M), L3 cache misses (L3M), energy consumption (NRG) and bytes read from (RD) and written to (WR) memory when executed in isolation.



(a) scikit-learn



(b) spec2006



(c) cloudsuite

Fig. 1. Absolute values of IPC and L3M distribution for different resource stressing scenarios. In all figures, x-axes and y-axes denote pairs of Last-Level Cache misses (L3M) and IPC as sampled from the PCM tool, respectively, whereas the line charts show the relocation of the distributions of L3M and IPC.

been revealed as memory bandwidth sensitive in prior works [11]. Finally, what is of great interest in the case of L3 stress scenario, is that, in spec2006 benchmarks the LLC misses increase, whereas in scikit-learn the misses decrease. This irregularity appears due to the combination of two factors; the performance degradation that the workloads experience due

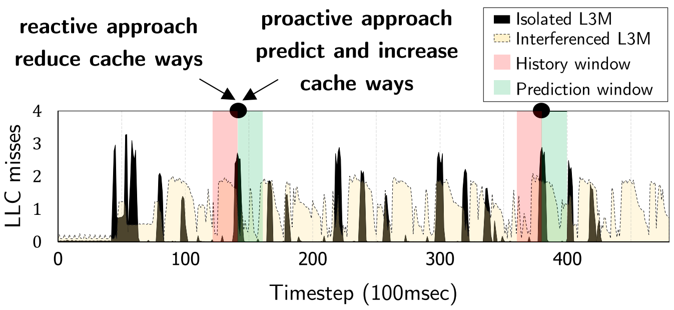


Fig. 2. Real trace of Random Forest Regressor benchmark showcasing the advantage of proactive predictive approaches compared to current state-of-the-art reactive approaches.

to interference and the sampling rate of the PCM tool. Performance degradation causes the signal traces to stretch, thus spreading the LLC misses in time. This reveals the importance of predictive monitoring in fine-grained timescales versus typical reactive performance monitors, since in cases of high application slowdown, even though the total number of LLC misses increase, a lower sampling rate would report lower LLC misses values, since the misses would be spread more widely in time.

3.4 Motivational Observations: Why Predictive Monitoring?

We showcase the importance of predictive monitoring with a typical example obtained throughout our experiments. Fig. 2 shows part of a real trace of LLC misses from the Random Forest Regressor benchmark. In this example, we assume that a runtime controller would take resource management decisions every 200msec (denoted as blue dotted lines). In addition, we suppose that the resource of interest is the number of cache ways given to the core executing the application. We give emphasis on the point highlighted with the black bullet. As we mentioned before, current state-of-the-art approaches [11], [30], [52] rely on reactive decision making approaches where resources are shared based on the *"history"* state of the system (red window in the figure). Since the LLC misses in this phase of the application fluctuate at low values, a reactive approach would result in lowering the cache-ways given to the application. However, we observe that in the upcoming window (green highlighted), the LLC misses of the application increase and, therefore, the number of cache-ways should be incremented as well. A proactive, predictive approach could determine this behavior before it even occurs, thus prohibiting of such inefficient configurations. Interestingly, we may also notice that the same "behavior" appears repeatedly throughout the lifetime of the application, showing that pure reactive approaches would make the same mistake again and again, probably leading to a catastrophic, for the application, management of resources.

A deeper look at Fig. 2 reveals the importance of the other major aspect of Rusty, i.e., the interference awareness in predictive monitoring. As depicted, we have super-imposed in Fig. 2 the signal of LLC misses when executing the Random Forest Regressor benchmark under interference (the yellow signal). As shown, interference effects heavily modify the signal structure/characteristics of the original isolated LLC misses signal. In this case, as shown, the magnitude of the LLC misses peaks have become smaller, but their duration

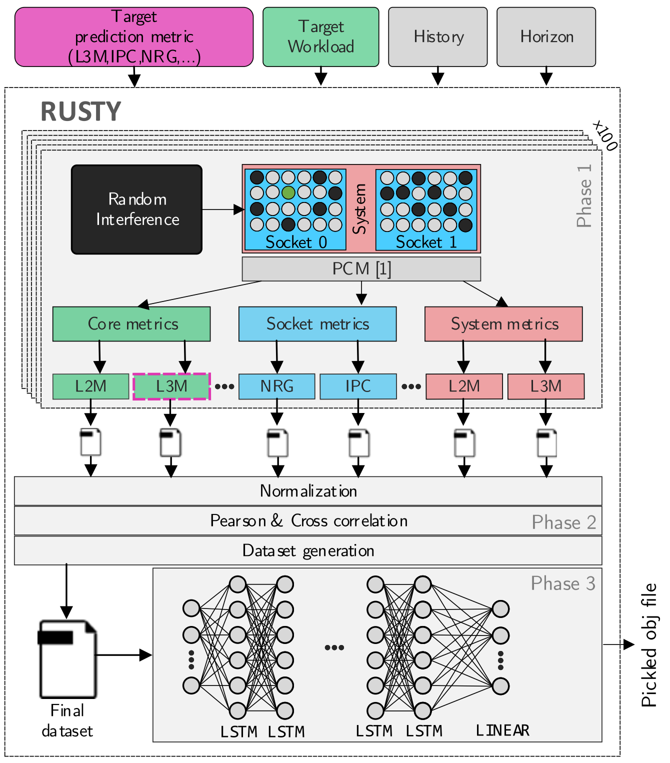


Fig. 3. Rusty's architecture overview.

have been increased. Focusing in the second black dot, around t=400, we may notice that while the reactive cache ways controller will fail for the same reasons as before, the same stands also for the case of the predictive controller that operates in an interference unaware manner, i.e., guided by a predictor trained only on isolated execution traces. More in detail, in this case, the interference unaware predictor will mis-predict resource demands, considering a steeper increase in L3 misses according to its isolated training set. However, as shown, the true L3 misses under interference presents a stable slope for the next period, thus imposing cache ways over-provisioning leading to either QoS degradation of co-running applications or in the worst case in resource starvation.

4 Rusty: Predictive Runtime Monitoring

Rusty relies on Long Short Term Memory networks [83], a special kind of Recurrent Neural Networks, that are able to make predictions based on long-term sequential dependencies. Instead of a single neural network layer, an LSTM cell contains four layers, three sigmoids (σ) and one hyperbolic tangent ($tanh$) layer. In brief, the first layer, also called "forget gate layer", applies a sigmoid function to the inputs and decides what information the network will throw away from the cell state. The second sigmoid layer is called the "input gate layer" and decides, along with the $tanh$ layer what new information is going to be stored and updated inside the cell. Finally, the last sigmoid layer acts as a refinement filter that decides which values are going to leave the cell.

4.1 Training Rusty

Fig. 3 shows an overview of the general procedure followed by Rusty to train the LSTM model. As shown, Rusty initially receives four inputs: i) the workload we would like to predict low-level metrics for (Section 3.3), ii) the performance metric to be predicted, which can be any metric from the PCM tool (for the current paper we focus on IPC, L3M and NRG - Section 3.2), iii) a variable called *history* and iv) a variable called *horizon*. *History* refers to the number of samples derived from the PCM tool and used as input sequence to the LSTM model, whereas *horizon* refers refers to the number of output features/future values that the LSTM model will predict. It should be noted here that, both *history* and *horizon* are relative to the sampling rate of PCM, i.e., for a sampling rate of 1 second, a *history/horizon* value of 15 would imply samples corresponding to 15 seconds of information, whereas for a sampling rate of 0.1 seconds, the same value would imply samples corresponding to 1.5 seconds. For the rest of the paper, we set the monitoring interval of PCM equal to 100msec. As shown in prior work [28], the monitoring interval can significantly affect the effectiveness of the model, with intervals greater than 100msec having negative impact on the accuracy.

Interference-Aware Trace Collection. The first phase of Rusty is responsible for collecting applications traces of our target workload under interference. To achieve this, the target application is executed on the system 100 times, each time with differing interference load, which can be either real or synthetic (see Section 4.3). For the rest of this analysis section, we imitate interference by utilizing the iBench suite [82], which provides contentious micro-benchmarks, where each one targets a different shared resource in a multi-core chip (processing cores, cache capacity, main memory bandwidth) and has tunable intensity. Specifically, we spawn random jobs from the iBench suite (up to the number of available threads) with various intensity levels, where each one is assigned on a randomly selected core of the system. During the execution, the framework collects and stores all information regarding performance metrics of the system, through the PCM tool. Following the above procedure, iBench produces static interference for the whole duration of the workload's lifetime. However, by considering multiple iBench co-execution scenarios per application, we note that the interference effects interference effects per scenario can change dramatically, thus producing disparate impact on the performance of the examined workload.

Data Pre-Process and Dataset Generation. The outcome of phase 1 is 100 low-level metric traces, which correspond to the different execution scenarios under varying interference. Phase 2 forms a pre-process phase, where the raw data of Phase 1 are prepared and processed in order to create the final dataset for our LSTM model. First, due to the different scales on the assembled data, we perform a min-max normalization, in order to bring the data values between the range [0, 1]. Then, we calculate some metrics of interest (pearson and cross correlation), which are used to minimize the amount of data fed to our model. We discuss this process in detail, in Section 4.2. The last step of Phase 2, is the dataset generation. Fig. 4 depicts the procedure followed to create our training and test datasets. To simulate the operation of a runtime monitoring tool, Rusty considers sliding windows that split the time-series derived through Phase 1 to parts of the same length. These parts correspond to different execution phases of the application, which, at inference time, will match to the respective runtime system metrics collected by Rusty.

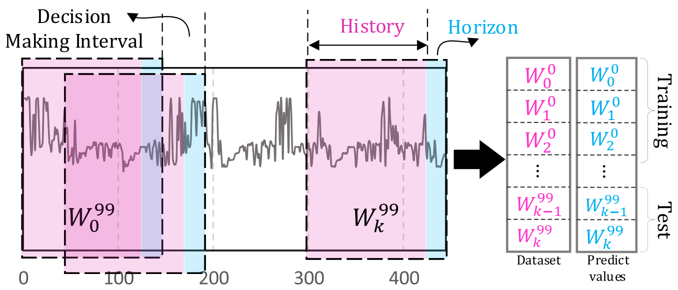


Fig. 4. Procedure followed to generate training and testing datasets. Each trace collected in Phase 1 is split into equal windows W with length $history + horizon$, each of which forms an instance of our final dataset.

Specifically, every sequence is divided into segments of length equal to $history + horizon$, each of which forms an instance of our final dataset. Also, the sliding value of the window corresponds to the decision making interval of a runtime resource manager. Without loss of generality, we suppose that a runtime manage rwould take resource management decisions every 2 seconds. Finally, all these windows form our final dataset, which is then split into training and test set accordingly.

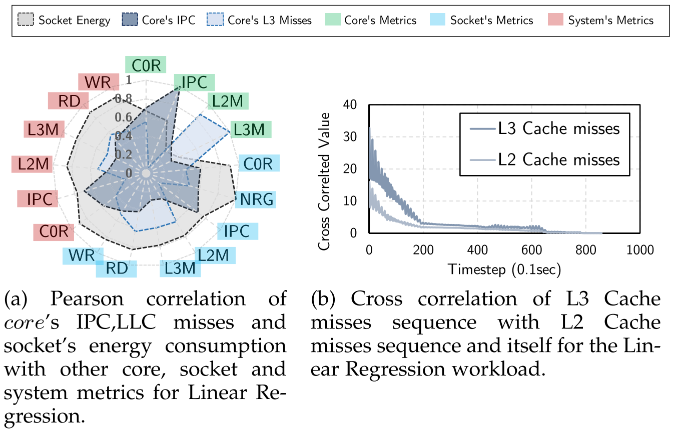
LSTM D.S.E. and Training. Finally, after the data have been pre-processed, and the respective dataset has been created, Phase 3 performs a Design Space Exploration (DSE) over the configurable parameters of the LSTM model, which we describe in detail, in Section 4.2. Once the overall best parameters are defined, the final LSTM model is trained and the learnable parameters (i.e., weights and biases) are saved to a pickled object file.

4.2 Rusty Architecture and Hyper-Parameter Tuning

The real-time requirements of runtime monitoring demands that the LSTM model should be as compact as possible, i.e., use as little features as possible and also, carry as little information as possible from the PCM tool for faster processing. In this section, we explore parameters that affect the accuracy and the depth of our model. Our goal is to provide an architecture that can achieve high accuracy, w.r.t. real-time constraints.

Which Metrics to Choose as Input Features?. For each one of the 100 executions performed in the offline part, certain metrics regarding the cores, sockets and the whole system are obtained. Then, Rusty designates the metrics to train the LSTM model with. To do so, it evaluates the correlation of the target prediction variable with the other performance metrics using the Pearson correlation coefficient.

Fig. 5a shows the correlation of IPC, L3M and NRG with the rest PCM's performance counters for the Linear Regression workload. The correlation pattern of the LR workload presents great similarities with the patterns exhibited among the others workloads, which are not presented due to space limitations. As shown, there is a high diversity between the correlations of the prediction variables (self-correlation equals to 1). NRG shows high correlation values with all the metrics related to the socket and the whole system, whereas it demonstrates lower correlation values with the core's metrics. As expected, L3 misses are highly correlated with L2 misses, as well as read (RD) and write (WR) operations from/to the memory. The correlation between L2 and L3 misses does not negate the fact that stressing the L2 cache does not impose degradation on the



(a) Pearson correlation of *core*'s IPC,LLC misses and socket's energy consumption with other core, socket and system metrics for Linear Regression.

(b) Cross correlation of L3 Cache misses sequence with L2 Cache misses sequence and itself for the Linear Regression workload.

Fig. 5. Correlation exploration for design space pruning.

performance of workloads. We expect L2 and L3 misses to follow a highly correlated pattern, since L2 misses consequently lead to higher traffic towards L3. In cases of interference, L2 misses are more likely going to lead to L3 misses, since more workloads contest for last level cache occupancy and, therefore, they continuously erase the contents of the cache. On the other hand, stressing the L2 cache does not inherently implies a large drop in performance, since we still have the chance to find our data in the LLC, thus, remaining on-chip.

Experimental evaluation of the accuracy of Rusty's 2-input LSTMs showed a slightly but consistent better accuracy when considering correlated metrics as model inputs. Thus, Rusty considers only the two most correlated metrics to train the LSTM model with.

How Far Back to Seek for Valuable Information? LSTMs capture dependencies on sequential data. Thus, an important design decision is determining the length of the sequence provided to our model (*History* value), i.e., how far back to search for valuable information. To extract this information, Rusty calculates the cross-correlation between the metrics provided as input features to the model. Fig. 5b shows the auto-recurrence degree found the L3 and L2 misses time series, under interference, for the Linear Regression benchmark. As shown, there is a reducing (as expected) effect, quite high though up to the first 100 points. In all benchmarks, we observed a reducing/decaying effect, quite high though, up to the first 50-70 points for small benchmarks and up to 200 points for larger benchmarks. In order to ensure effective auto-correlation information, Rusty focuses on sequence lengths between 50-70 points, covering the region exhibiting the highest recurrence dependencies. However, we note that Rusty supports configurable "history" windows, easily adapted to differing "history" sizes.

Fig. 6 provides insights regarding the R^2 achieved for different *History-Horizon* combinations. Specifically, each cell of the heatmap illustrates the geometrical mean of R^2 scores for the respective, metric, *history* and *horizon*, over all our target workloads. As shown, the LSTM network can effectively predict L3M and IPC for a *horizon* of $25-35$ points, given as input *history* sequences of 12 samples or more. Regarding the socket's energy consumption, we observe that our model can predict equivalent *horizon* windows, even with a single *history* value, showing the extensive expressiveness of LSTMs on low rank fluctuating signals. In the examined case of forecasting during the execution of a

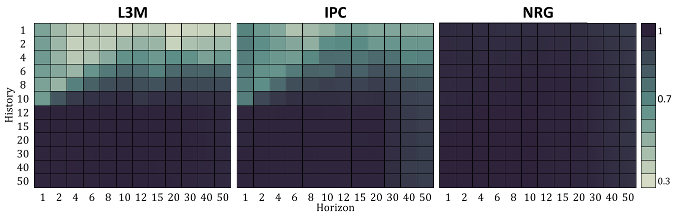


Fig. 6. Geomean of *History* versus *Horizon* accuracy for the LSTM model over all workloads.

specific mixture of co-running workloads, energy signals can be safely consider to expose low rank fluctuations, since the major power consumption coefficients are more due to the activation of specific cores, cache memories etc and less due to IPC and L3 misses runtime variances.

How Complex Should the LSTM Architecture Be? Since Rusty's design should be as lightweight as possible, we explore and evaluate three different parameters that affect the complexity and precision of the LSTM model, particularly, the numbers of stacked LSTM layers, the number of features per layer and the number of epochs used during training. We use the benchmarks from the scikit suite for this exploration.

Fig. 7 illustrates three boxplot diagrams, showing the effect and the corresponding robustness of each examined parameter, w.r.t. the accuracy of the model, assessed through the coefficient of determination - R^2 score. As shown, even though the number of epochs does not affect the complexity of the model, it plays a crucial role in its accuracy, since higher number of epochs increase the precision of the model dramatically, regardless the other design characteristics. In addition, increasing the number of features positively affects the accuracy up to a plateau, after which a significant degradation is observed. The same is true for the number of stacked LSTM layers. Stacking LSTM hidden layers makes the model deeper, thus making the network more eager to recognize certain aspects of input data. Since LSTMs operate on sequencial data, the addition of layers allows the hidden state at each level to operate at different timescale [84]. However, adding frantically more layers can degrade performance of the network. Once the network starts converging, adding more layers can cause its accuracy to get saturated and then degrade rapidly [85].

After filtering out the inefficient configurations, e.g., LSTM layers > 4, we further attempt to determine a single "golden" LSTM architecture, (i.e., the number of layers and features, not the coefficient values of a trained model) that delivers optimized prediction capabilities across the majority of the target workloads. In order to capture the central tendency, we computed the geometric mean of R^2 scores over the various application-specific trained LSTMs configurations of the scikit workloads. As shown in Fig. 8, despite the fact that all exam-

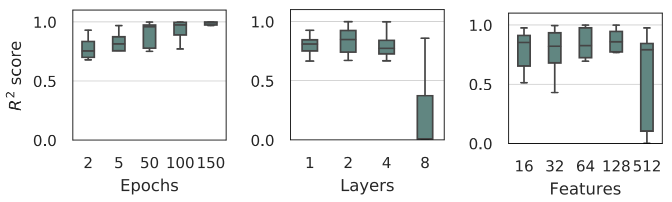


Fig. 7. Exploration on the impact of different design pa-rameters on the overall accuracy of the model.

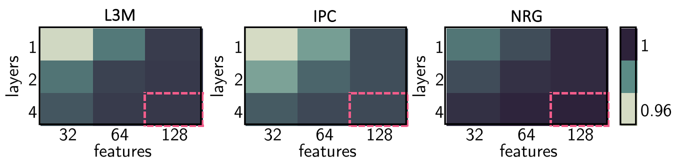


Fig. 8. Geometric mean of L3M, IPC, and NRG R^2 scores.

ined LSTM architectures deliver high geomean R^2 accuracies, i.e., over 0.96, the stacked LSTM network with 4 layers and 128 features per layer outperforms all the other configurations for all the addressed performance metrics.

4.3 How to Deploy Rusty

Rusty is as an intra-node monitoring system, that predicts future low-level system metrics and, thus, can be replicated across multiple nodes of a cluster to form a realistic solution in a scale-out cluster configuration. Rusty has been integrated with the PCM tool, through `Python` and `C++` embedding, in order to provide runtime predictions, without the need of intermediate logs and files. In addition, all the measured and predicted metrics, from PCM and Rusty respectively, are stored in a `MySQL` [80] database, thus, allowing the integration of our framework with advanced visualization and workload management tools, such as Prometheus [42] or Grafana [86].

Rusty can be utilized under several deployment scenarios, either cloud-native or VM-based from both a cloud-user or cloud-provider perspective. Its only restriction is that of requiring access to model-specific registers (MSR) to monitor low-level system metrics. While historically VMs didn't allow information on low performance counters to be available, recent advances like VMware's virtualized Model-Specific Registers (MSRs) in vSphere technology establish such observability. However, recently VMs are not the only option available for Infrastructure-as-a-Service purchase. Cloud providers made possible to rent bare-metal, physical machines, such as the m5.metal instances on AWS, which were also used later in our evaluation Section 5.3. In such cases, Rusty can be deployed without any further requirements to provide runtime predictive monitoring on workloads running on the system. From a *cloud-provider* perspective, Rusty can form a advance solution to provide predictive monitoring of the performance of VMs and aid the infrastructure devops so that to enable better Quality of Service to their customers.

Cluster Deployment. Fig. 9 shows how Rusty can be utilized on a cluster of multiple nodes, where the cluster is managed by a master orchestrator (such as Kubernetes) and there is an additional controller on each node of the cluster, which manages resources inside the system according to system state predictions accepted through Rusty. In such a setup, there is a Rusty instance running at each node, responsible for monitoring and making future predictions regarding the specific system's metrics. Specifically, Rusty continuously collects PCM metrics every 100 msec and makes future predictions at each decision making interval (as described in Section 4.1).

These metrics are used by both the master/inter-node controller as well as the intra-node controller for decision making regarding management of resources. The inter-node controller leverages such information for more efficient initial placement

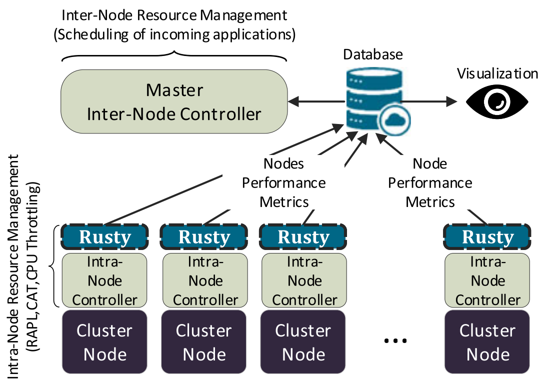


Fig. 9. Example of Rusty's integration on a cluster of multiple nodes. Rusty can be integrated with both a potential intra-node controller, to provide per node resource management decisions, as well as an inter-node for better cluster orchestration.

of newly arriving workloads on the cluster, whereas, the intra-node controller takes advantage of the predictions, to proactively manage resources on the system (e.g., perform dynamic power capping or alter the cache allocation scheme).

Rusty's Training Modes. Typically, Rusty requires no a priori knowledge of the workloads running on the system. Rusty can be deployed in two ways, either with pre-trained models (pickled object files), which have been trained offline, or in a *stripped*-manner, where the models are trained online, dynamically. This first way covers occasions where custom, personalized models, are necessary, offering higher accuracy for deeper *horizons*. However, this comes with the drawback of not perceiving the real interference the target system might experience. In such situations, interference can be emulated by utilizing synthetic micro-benchmarks for imitating interference, like iBench [82].

The second way covers occasions, where the LSTM models are trained with real, interference-aware traces, derived directly from the system that Rusty has been deployed to. In such circumstances, Rusty first experiences a *grace period*, during which it collects low-level performance counters from the workloads executing on the system. In a scale-out cluster, where multiple machines execute the same workloads under diverse interference scenarios, the *grace period* can be completed in orders of hours. After the metrics have been collected, Rusty can train either workload-specific models, responsible for more fine-grained predictions, possibly targeting LC workloads, or general, coarse-grained ones, covering a wider range of applications (we show how Rusty extrapolates to new workloads in Section 5.2).

Adapting to Abnormalities. Rusty continuously monitors and evaluates the predicted metrics by comparing them with the actual ones. If new workloads arrive on the system, they may expose undiscovered interference effects that have not been revealed during the Rusty's training phase. If a deviation in the prediction efficiency is discovered, then Rusty retrains the model in order to adapt and update the weights and the biases of the model accordingly.

5 Evaluation

In this section, we explore Rusty's efficiency utilizing well-established techniques from the machine learning and data

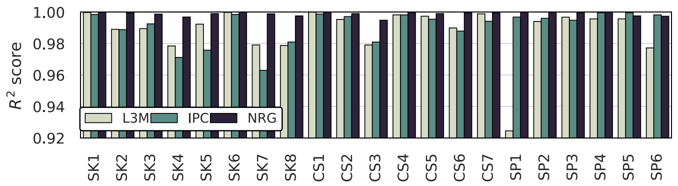


Fig. 10. L3M, IPC, and NRG R^2 score per benchmark for the overall best configuration (4 stacked LSTM layers and 128 features/layer), *history = 50* and *horizon = 1*.

engineering domains to validate its efficiency and to enable comparative and reproducible studies w.r.t other SoA solutions. Specifically, we generated several interference scenarios, where each scenario is constituted by a different random number of spawned workloads at random times. e note that the following experimental results are based on workload co-locations as derived from mixing applications found in [35], [36], [37]. We evaluate Rusty on real continuous predictive monitoring deployments, i.e., considering very fine grained time-resolutions in comparison with other LSTM-based based approaches, e.g., Seer [28]. By this way we are showing the robustness of Rusty concepts on realistic/pragmatic interference scenarios, in comparison with Sections 3.3 and 4, where we considered synthetic microbenchmarks from the iBench suite for our analysis.

We partition the datasets produced during execution in two subsets of 90 percent (training set) and 10 percent (test set) of the samples respectively. The LSTM models were trained using the PyTorch library. The batch size of our dataset was set equal to 64, the learning rate equal to 0.001 and we utilized ADAM as our optimizer. Furthermore, all the models were trained using the architecture obtained from Section 4.2 (4 stacked LSTM layers and 128 features per layer).

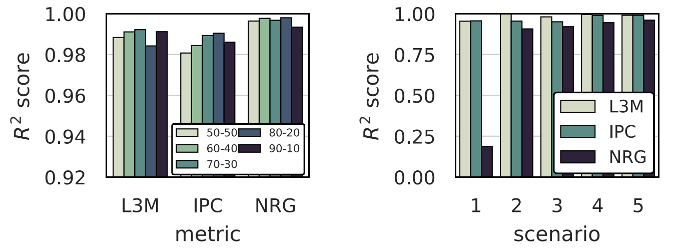
5.1 Rusty's Accuracy

Fig. 10 shows the accuracy of Rusty for predicting L3M, IPC and NRG per workload, for a *history* value equal to 50 and *horizon* equal to 1. As we see, Rusty achieves pretty high accuracies from 0.92 up to 0.99 R^2 scores for all the three target prediction variables, with 0.991, 0.988 and 0.994 R^2 score on average for L3M, IPC and NRG, respectively. In addition, it is notable that the considered LSTM architecture, although extracted over the scikit suite, also fits and exposes high predictability on the cloudsuite and spec2006 workloads.

To further validate the robustness of Rusty, we also examine the accuracy for different splits of training and test sets. Fig. 11a depicts the average accuracy among all target workloads for training-test split percentages equal to 50-50, 60-40, 70-30, 80-20 and 90-10 of the whole dataset. As shown, Rusty consistently exhibits high accuracies for all our metrics of interest, over different training-test splits, ranging from 0.98 up to 0.998. This forms a very important and promising outcome, showing that Rusty can deliver very high prediction capabilities without resorting to extremely time consuming traversals on the overall LSTM design space each time a new workload is given as input.

5.2 How Does Rusty Behave on Unseen Workloads?

In order to investigate the ability of Rusty to extrapolate to new workloads, we generated and evaluated 5 different



(a) Rusty's accuracy for different train-test dataset splits.

(b) Rusty's ability to extrapolate to new workloads

Fig. 11. Rusty's flexibility on training datasets and workloads.

scenarios, regarding the workloads used during training and inference, which are described in Table 3. Specifically, for the first 3 scenarios, all but one suites are used as training sets and the remaining one as testing. Since each benchmark suite captures specific system utilization patterns, we note that the aforementioned scenarios define a set of extremely stressed prediction cases, where the extracted model is called to make inference on unseen patterns, i.e., patterns not trained on. Respectively, in scenarios 4 and 5, the first half of the workloads from each suite are used as training set and the second half as inference, and vice-versa.

Fig. 11b depicts the R^2 achieved for all the target prediction variables and for each one of the aforementioned scenarios. This figure also reveals the capability of Rusty to achieve high prediction efficiency as far as the IPC and the L3M metrics are concerned, with an average of 0.965 and 0.988 R^2 scores respectively, among all scenarios. Regarding the NRG metric, we can see that Rusty exhibits high accuracies for scenarios 2-5, but experiences a huge drop on scenario 1. This inaccuracy can be interpreted if we take a close look at Table 2. In scenario 1, we used workloads from the scikit-learn and the spec2006 suites as our training dataset and cloudsuite as our test one. Regarding the former workloads (train set) we can observe that the ratio of NRG to IPC is approximately in the range (2,3), whereas for the latter ones (test set) the ratio increases to (5,6). Therefore, in scenario 1, we are enforcing the LSTM network to make predictions on patterns either under- or non-represented in our training data, i.e., reassembling an inefficient/problematic data engineering strategy where careless selection of training set fail to paint the whole picture.

5.3 How Does Rusty Behave on Unseen Machines?

We further investigate whether Rusty can be directly transferred, i.e., without re-training, among machines with different specifications and micro-architectures. This forms also an extreme stressed evaluation scenario for Rusty, to evaluate its generalization capabilities without adapting the

TABLE 3
Different Training/Inference Scenarios

#	Training	Test
1	SK[1-8], SP[1-6]	CS[1-7]
2	SP[1-6], CS[1-7]	SK[1-8]
3	CS[1-7], SK[1-8]	SP[1-6]
4	SK[1-4], SP[1-3], CS[1-3]	SK[5-8], SP[4-6], CS[4-7]
5	SK[5-8], SP[4-6], CS[4-7]	SK[1-4], SP[1-3], CS[1-3]

TABLE 4
AWS EC2 m5.metal Specifications

Processor Model	Intel® Xeon® Platinum 8175M
Cores per socket	24 (48 logical) @2.50GHz
Sockets	2
L1 Cache	32KB instr. & 32KB data
L2 Cache	1024K
L3 Cache	33MB, 11-way set-associative
Memory	396GB @2666 MT/s
Operating System	Ubuntu 18.04, kernel v4.15

typical full-retraining approach used in other predictive model solutions [28].

We deploy Rusty on a baremetal, m5.metal, machine (to allow access to MSR registers) from Amazon EC2 [3] (Table 4). We execute 100 different scenarios, where, at each scenario, a random number of workloads are spawned from the pool of spec2006 and scikit-learn suites. We utilized the Rusty model trained with traces derived from our initial server system (Table 1) and evaluate the accuracy of this model on traces from the EC2 instance.

Fig. 12 shows the R^2 achieved per workload for all the three target metrics, for a *history* value of 50 and a *horizon* of 20. As shown, Rusty experiences a slightly lower, but still high, accuracy, with R^2 scores ranging from 0.76 up to 0.99, experiencing an average reduction of 0.01, 0.10 and 0.12 for L3M, IPC and NRG respectively. This experiment reveals that, even though the underlying architecture has changed, there are repeated patterns in the signal traces of the workloads, which Rusty is able to capture, due to the normalization performed during the data pre-process phase. However, to be able to determine the real values of the metrics, we should maintain the max value of each metric used during normalization. Moreover, the accuracy of the model can be further improved by either retraining the network or by applying transfer learning techniques [87]. Nevertheless, this figure reveals the ability of Rusty to extrapolate to new machines, thus forming an ideal monitoring solution for modern data-centers that feature server nodes with differing micro-architecture.

We note that the aforementioned experimental study examined a limited of differing architectures, showcasing as a proof-of-concept Rusty's generalization capabilities. This does not mean that Rusty is transferable retaining its high efficiency among all different architectures available on a data-center without any modifications and/or retraining.

5.4 Comparative Analysis

To ascertain the superiority of LSTMs over simpler ML models and time-series analysis techniques, we further perform a comparative analysis between the accuracy of our adopted LSTM versus i) an Autoregressive Integrated Moving Average (ARIMA) model, ii) a Linear Regression (LR) model and iii) a Multi-Layer Perceptron Regressor (MLP). LR has been used in prior works for predicting system and higher-level metrics [46], [88], whereas MLP was chosen due to its ability to capture non-linear dependencies. For the ARIMA technique we utilized the `auto_arima` function from python's `pyramid.arima` sub-module, where as the LR and MLP models were trained with the `scikit-learn` [35] library. We examine the accuracy of the aforementioned models, trained for differing *history* window sizes as well as prediction *horizons*, both ranging between [4, 50], under various random interference scenarios, i.e., considering randomly co-located workloads from scikit, cloudsuite and spec2006 suites.

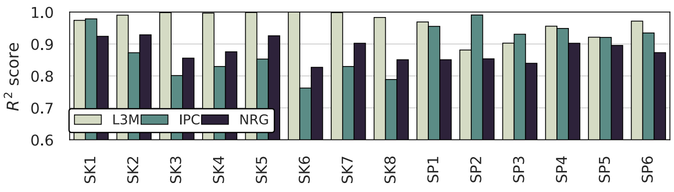


Fig. 12. Rusty's ability to extrapolate to new machines.

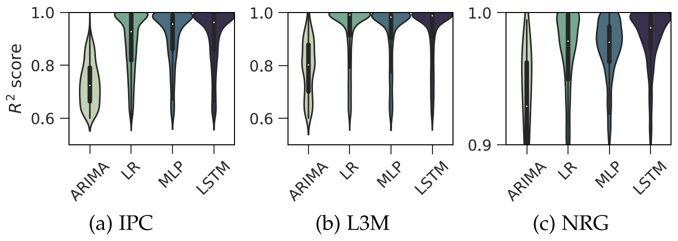


Fig. 13. R^2 score distribution of ARIMA, Linear Regression, Multi-Layer Perceptron ARIMA and LSTM over all workloads for history-horizon pairs ranging in [4, 50].

Fig. 13 shows a violin-plot diagram comparing the four examined models, over all our target workloads and metrics, by considering the distribution of R^2 scores achieved. As shown, LSTM outperforms the other three approaches, providing more robust and accurate predictions, since for all the three examined metrics depicted both the median as well as the 25th and 75th percentiles values of the respective distributions are greater. Moreover, the converged violin shape of the LSTM closer to R^2 scores equal to 1 reveals that even though LR and MLP also provide quite accurate predictions, LSTM tends to behave better providing consistently better forecasting of future metrics. Finally, throughout our experiments we observed that, for deeper horizon values (> 6), the score gap between LR and LSTM rises constantly, showing the deficiency of the linear model to capture non-linearities in the time-series signal, whereas MLP and LSTM behave almost the same, with LSTM being slightly more accurate in the majority of the cases.

5.5 Evaluating Rusty for Further PCM Metrics

As mentioned in Section 4.1, Rusty receives as input the low-level metric to be predicted. In this experiment, we evaluate Rusty for additional low-level performance counters provided by the PCM tool. Specifically, we assess the efficiency of Rusty for predicting the LLC occupancy (L3OCC) and the C0-state (C0) of the core executing our target workload, the reads(RD)/writes(WR) of the sockets of the system from/to the memory and also the total Quick Path Interconnect (QPI) traffic of the system for a history of 50 and an horizon of 20 samples, using the overall best architecture obtained from Section 4.2 and a train-test split of 70-30 percent respectively.

Table 5 shows the accuracy in terms of R^2 score achieved by Rusty for the aforementioned additional low-level

TABLE 5
Rusty's Accuracy for Additional PCM Metrics

L3OCC	C0-state	READ	WRITE	QPI
0.988	0.984	0.999	0.998	0.999

metrics. From this table it is evident that Rusty's methodology is not bound to the explicit low-level performance counters considered throughout the paper, but can be applied to any metric found in the system. Especially for metrics higher in the system hierarchy (e.g., socket or system metrics) Rusty can provide extremely accurate results, as shown both by the previous evaluation of the socket's energy consumption and also from the accuracy achieved for READs, WRITEs and the QPI traffic. The above analysis shows that Rusty forms an effective universal predictive framework for low-level system metrics, capturing interference from either aggregated traces, e.g., energy, reads or writes etc., found higher in the system hierarchy, up to more primitive ones, closer to the core level, e.g., per-core IPC, L3-occupancy etc.

5.6 Rusty's Overhead

Rusty's high frequency monitoring and inference engine inevitably affects the performance of the underlying system. Fig. 14 provides two heatmaps, showing the average CPU utilization as derived from Unix's `top` command, as well as the average inference time required for Rusty to predict the IPC for different number of OpenMP threads given for inference and cores for which Rusty predicts low-level metrics for. We should note here, that, by default, `top` displays CPU% as a percentage of a single CPU, of the system, thus, on multi-core we can have percentages that are greater than 100 percent.

These figures reveal that scaling the number of OpenMP threads does not reduce the inference time needed to predict future metrics, while at the same time infícts huge utilization overhead on our system. This is due to the fact that the overhead of dispatching the computation to multiple threads is not counterbalanced by the overall effort required to make the predictions for a single-vector inference batch. In addition, we also see that the inference time needed to make more than 4 predictions on the system cannot follow the high frequency monitoring interval of 100msec. In fact, Rusty requires almost 1 second to predict the IPC for all the 48 available cores on the system. From the above analysis, it is shown that a reasonable strategy regarding Rusty's placement would be to allocate a

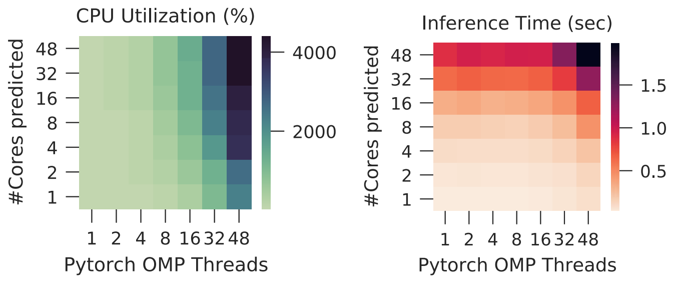


Fig. 14. Rusty's CPU utilization overhead and inference time for different number of system cores predicted and OpenMP threads given to Pytorch library.

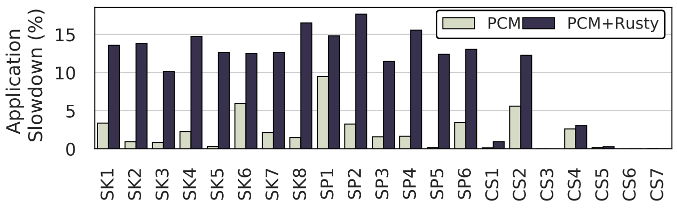


Fig. 15. PCM and Rusty performance overhead imposed per examined workload.

Rusty monitor in 1 core dedicating around 60 percent of its resources to predict metrics from 4 other cores of the system. We note that this level resource consumption is not expected to incur any starvation issues in real environments, since recent reports show that the average resource utilization inside data-centers is around 50 percent-60 percent [67].

To further evaluate the overhead by Rusty, we also provide the performance degradation imposed to our examined workloads when co-located on the same physical core with PCM-only and Rusty compared to a totally isolated execution. Fig. 15 shows the per-application slowdown (%). As we see most of the cloudsuite workloads are immune to interference effects imposed by the monitoring tools. The highest slowdown is experienced by application CS2, which uses Apache Spark to run a collaborative filtering algorithm. For all the other cases, we observe a similar pattern on the performance overhead of both PCM and Rusty, with the former imposing an average of 4 percent and the latter an average of 13 percent performance overhead respectively.

Finally, for completeness, we also provide the time for changing the cache allocation knob, i.e., CAT policy, and the power capping, using RAPL, of the targeted Intel Xeon server, which is 0.018s and 0.012s on average respectively. Based on the above, we can conclude that the imposed overhead of the proposed methodology is suitable for supporting online decision making and resource allocation.

5.7 Correlating Lower- With Higher-Level Performance Metrics

Predictive monitoring on performance counters provide low level information regarding the state of the underlying system at a per-resource manner, thus allowing us to detect true causes of interference and bottlenecks, i.e., which resource is responsible for the performance degradation our system (and consequently our workloads) is experiencing. However, it is also considered of great importance, these low level insights to be projected to higher-level metrics of interest, such as the slowdown applications experience, upcoming QoS violations etc. In [28], the authors showed that predictive classification of QoS violations is possible by exploiting the raw low-level performance counters. However, gaining more insight, e.g., accurately predicting the actual slowdown of a workload is a much more difficult regression problem, which requires additional models to be applied on top of them.

We showcase the importance but also the applicability of exploiting this low level monitoring information for slowdown predictability, through a rather simple exemplary scenario of training a Multi-Linear Perceptron (MLP) regressor that receives low-level metrics of the workload executed

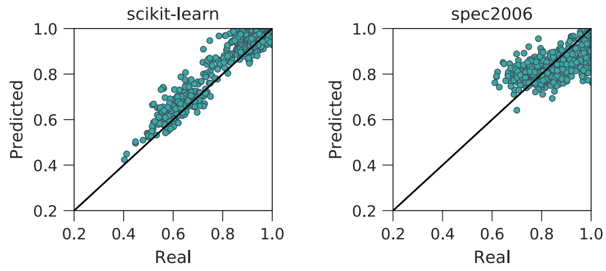


Fig. 16. Slowdown prediction residuals.

under interference as input features and predicts the respective slowdown experienced. To determine the primary inputs of our model, we evaluated the correlation between application slowdown, defined in this case as $\frac{T_{isolated}}{T_{interfered}}$, and various performance monitoring distribution statistics, and utilize the most highly correlated features, namely the degradation of the mean and median IPC, L3M and LLC occupancy compared to the isolated execution. Our model consists of 6 layers and 64 features per layer. Moreover, our dataset consists of 100 instances per suite's workload, each instance corresponding to execution with diverse interference, where 90 percent of the dataset is used as training and 10 percent as test set respectively. We also performed a *6-fold* cross-validation to examine the robustness of the model. Fig. 16 illustrates the residuals of the regression for the scikit-learn and cloudsuite benchmarks, where the x and y axes, show the real and predictive slowdown the application experienced. As demonstrated, the MLP predictor can quite effectively forecast the slowdown the workloads experienced, with a Mean Squared Error (MSE) of 0.003 for the scikit-learn and 0.03 for the spec2006 suite.[1] Although rather simple in implementation complexity, this exemplary case shows the potential of exposing low-level performance counters to higher-level metrics of interest, forming a promising solution for runtime control using low-level metrics.

6 Conclusion

Run-time predictability of systems under interference is essential for better management of resources in modern large-scale cloud data centers. We proposed Rusty, a lightweight LSTM-based predictive monitoring framework able to provide fast, accurate, non-intermittent and interference-aware predictions of low-level system metrics in multi-tenant systems. We analyzed and explored several schemes of LSTMs concluding to a generic efficient LSTM architecture in terms of accuracy, responsiveness to run-time constraints and computational cost. We showed that Rusty forms a realistic solution achieving extremely high prediction accuracy R^2 of 0.98 on average under pragmatic workload consolidation scenarios driving modern cloud platforms and also that it satisfies the strict latency constraints imposed by low-level system knob activation. We foresee Rusty to be integrated in existing orchestration frameworks, capturing complex system dynamics and assisting towards more elaborated resource allocation decisions.

1. For Cloudsuite, the MLP regressor did not provide the similar accuracy results, thus, more descriptive and deep networks might be required, which, however, are out of the scope of this paper.

Acknowledgments

This work has been supported in part by the European Union's Horizon 2020 Research and Innovation programme, EVOLVE under Grant 825061. The authors would like to thank the anonymous reviewers for their valuable comments, which improved the quality of this article. In addition, the authors would like to thank Dr. Georgios Retsinas and PhD candidate Athena Elafrou for their helpful insights and comments regarding the manuscript.

References

[1] C. V. Networking Index , "Forecast and methodology, 2016–2021, "White Paper, San Jose, CA, USA, vol. 1, 2016.
[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Queue*, vol. 14, no. 1, pp. 70–93, 2016.
[3] Amazon Web Services, Accessed: Aug. 8, 2020. [Online]. Available: https://aws.amazon.com/
[4] Google Cloud Platform, Accessed: Aug. 8, 2020. [Online]. Available: https://cloud.google.com/
[5] Microsoft Azure Cloud Computing Platform & Services, Accessed: Aug. 8, 2020. [Online]. Available: https://azure.microsoft.com
[6] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 77–88, 2013.
[7] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 450–462, 2015.
[8] 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 *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 248–259.
[9] H. Zhu and M. Erez, "Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 2, pp. 33–47, 2016.
[10] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars, "Enabling fair pricing on high performance computer systems with node sharing," *Sci. Program.*, vol. 22, no. 2, pp. 59–74, 2014.
[11] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. 24th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2019, pp. 107–120.
[12] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, Art. no. 51.
[13] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "GrandSLAm: Guaranteeing SLAs for jobs in microservices execution frameworks," in *Proc. 14th EuroSys Conf.*, 2019, Art. no. 34.
[14] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, vol. 30, no. 4, pp. 8–19, Jul./Aug. 2010.
[15] S. Kanev *et al.*, "Profiling a warehouse-scale computer," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3, pp. 158–169, 2016.
[16] Instana - APM for Dynamic Applications, Accessed: Aug. 8, 2020. [Online]. Available: https://www.instana.com/
[17] K. Du Bois , S. Eyerman, and L. Eeckhout, "Per-thread cycle accounting in multicore processors," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 29:1–29:22, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400688
[18] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. 15th Edition ASPLOS Architectural Support Program. Lang. Operating Syst.*, 2010, pp. 335–346. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736058
[19] L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proc. 48th Int. Symp. Microarchit.*, 2015, pp. 62–75. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830803

[20] A. Herdrich *et al.*, "Cache QoS: From concept to reality in the Intel® Xeon® Processor E5–2600 v3 product family," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 657–668.

[21] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proc. 16th ACM/IEEE Int. Symp. Low Power Electron. Des.*, 2010, pp. 189–194.

[22] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, Art. no. 217.

[23] C. Isci and M. Martonosi, "Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques," in *Proc. 12th Int. Symp. High-Perform. Comput. Archit.*, 2006, pp. 121–132. [Online]. Available: https://doi.org/10.1109/HPCA.2006.1598119

[24] 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, pp. 22:1–22:25, Jun. 2019. [Online]. Available: http://doi.acm.org/10.1145/3323090

[25] Dynatrace Official Website, Accessed: Aug. 8, 2020. [Online]. Available: https://www.dynatrace.com/

[26] ITRS Group: For the always-on enterprise, Accessed: Aug. 8, 2020. [Online]. Available: https://www.itrsgroup.com/

[27] Splunk official website, Accessed: Aug. 8, 2020. [Online]. Available: https://www.splunk.com/

[28] Y. Gan *et al.*, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 19–33, doi: 10.1145/3297858.3304004

[29] X. Yu, P. Joshi, J. Xu, G. Jin, H. Zhang, and G. Jiang, "Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 489–502, 2016.

[30] Y. Sfakianakis, C. Kozanitis, C. Kozyrakis, and A. Bilas, "QuMan: Profile-based improvement of cluster utilization," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, 2018, Art. no. 27.

[31] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *Proc. 24th ACM Int. Conf. Supercomputing*, 2010, pp. 147–158.

[32] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *Proc. USENIX Annu. Tech. Conf.*, 2011, p. 12.

[33] M. Zaman, A. Ahmadi, and Y. Makris, "Workload characterization and prediction: A pathway to reliable multi-core systems," in *Proc. IEEE 21st Int. On-Line Testing Symp.*, 2015, pp. 116–121.

[34] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online QoS management for increased utilization in warehouse scale computers," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 607–618, 2013.

[35] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.

[36] M. Ferdman *et al.*, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. 17th Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2012, pp. 37–48, doi: 10.1145/2150976.2150982.

[37] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006.

[38] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 22–22.

[39] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: Approaches, applications, and evolutions," *ACM Comput. Surv.*, vol. 48, no. 1, 2015, Art. no. 10.

[40] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*. Berlin, Germany: Springer, 2010, pp. 157–173.

[41] T. Willhalm, R. Dementiev, and P. Fay, "Intel performance counter monitor-a better way to measure CPU utilization," *Dosegljivo*, 2012. [Online]. Available: https://software.intel.com/en-us/articles/intelperformance-counter-monitor-a-better-way-to-measure- cpu-utilization

[42] *Prometheus—Monitoring System & Time Series Database*, Accessed: Aug. 8, 2020. [Online]. Available: https://prometheus.io/

[43] J. Varia and S. Mathew, "Overview of amazon web services," *Amazon Web Services*, pp. 1–22, 2014.

[44] R. Sen and D. A. Wood, "Reuse-based online models for caches," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 279–292, 2013.

[45] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 127–144, 2014.

[46] N. Mishra, J. D. Lafferty, and H. Hoffmann, "ESP: A machine learning approach to predicting application interference," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2017, pp. 125–134.

[47] R. Xu *et al.*, "PYThHIA: Improving datacenter utilization via precise contention prediction for multiple co-located workloads," in *Proc. 19th Int. Middleware Conf.*, 2018, pp. 146–160.

[48] Y. Li *et al.*, "GAugur: Quantifying performance interference of colocated games for improving resource utilization in cloud gaming," in *Proc. 28th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 231–242. [Online]. Available: http://doi.acm.org/10.1145/3307681.3325409

[49] N. Morris, C. Stewart, L. Chen, R. Birke, and J. Kelley, "Model-driven computational sprinting," in *Proc. 13th EuroSys Conf.*, 2018, Art. no. 38.

[50] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *Proc. 13th USENIX Symp. Netw. Syst. Des. Implementation*, 2016, pp. 363–378.

[51] Y. D. Barve *et al.*, "FECBench: A holistic interference-aware approach for application performance modeling," in *Proc. IEEE Int. Conf. Cloud Engineering (IC2E)*, 2019, pp. 211–221.

[52] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for QoS-aware clouds," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 237–250.

[53] S. Blagodurov and A. Fedorova, "User-level scheduling on numa multicore systems under linux," in *Proc. Linux Symp.*, 2011, pp. 81–91.

[54] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *Proc. IEEE Int. Symp. Workload Characterization*, 2013, pp. 66–76.

[55] L. Wang *et al.*, "BigDataBench: A big data benchmark suite from internet services," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit.*, 2014, pp. 488–499.

[56] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *Proc. IEEE Int. Symp. Workload Characterization*, 2014, pp. 202–211.

[57] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "Counterminer: Mining big performance data from hardware counters," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2018, pp. 613–626.

[58] M. Du and F. Li, "ATOM: Efficient tracking, monitoring, and orchestration of cloud resources," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2172–2189, Aug. 2017.

[59] M. G. Moghaddam, W. Guan, and C. Ababei, "Investigation of LSTM based prediction for dynamic energy management in chip multiprocessors," in *Proc. 8th Int. Green Sustain. Comput. Conf.*, 2017, pp. 1–8.

[60] M. Hashemi *et al.*, "Learning memory access patterns," in *Proc. 35th Int. Conf. Mach. Learn.*, G. D. Jennifer and K. Andreas, Eds., 2018, vol. 80, pp. 1924–1933. [Online]. Available: http://proceedings.mlr.press/v80/hashemi18a.html

[61] *Processor Counter Monitor (PCM)*, Accessed: Aug. 8, 2020. [Online]. Available: https://github.com/opcm/pcm

[62] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[63] J. Hauswald *et al.*, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 223–238, 2015.

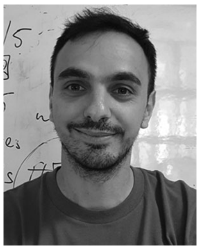
[64] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Proc. 19th Int. Conf. Parallel Archit.s Compilation Techn.*, 2010, pp. 557–558.

[65] G. Ayers *et al.*, "AsmDB: Understanding and mitigating front-end stalls in warehouse-scale computers," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 462–473. [Online]. Available: https://doi.org/10.1145/3307650.3322234

[66] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 152–162. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155638

[67] J. Guo *et al.*, "Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces," in *Proc. Int. Symp. Quality Service*, 2019, Art. no. 39.

[68] S. Blagodurov, S. Zhuravlev, and A. Fedorova, “Contention-aware scheduling on multicore systems,” *ACM Trans. Comput. Syst.*, vol. 28, no. 4, 2010, Art. no. 8.
[69] Y. Liu, S. C. Draper, and N. S. Kim, “Sleepscale: Runtime joint speed scaling and sleep states management for power efficient data centers,” in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, 2014, pp. 313–324.
[70] Q. Wu *et al.*, “Dynamo: Facebook’s data center-wide power management system,” in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 469–480.
[71] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, “Towards energy proportionality for large-scale latency-critical workloads,” in *Proc. 41st Annu. Int. Symp. Comput. Architecuture*, 2014, pp. 301–312. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665718
[72] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, “Pack & cap: Adaptive DVFS and thread packing under power caps,” in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2011, pp. 175–185. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155641
[73] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, “PPEP: Online performance, power, and energy prediction framework and FVFs space exploration,” in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2014, pp. 445–457. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.17
[74] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
[75] A. Cassandra, “Apache cassandra,” 2014. Art. no. 13. [Online]. Available: http://planetcassandra.org/what-is-apache-cassandra
[76] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *Proc. 2nd USENIX Conf. Hot Topics Cloud Comput.*, 2010, Art. no. 95.
[77] NGINX official website, Accessed: Aug. 8, 2020. [Online]. Available: https://www.nginx.com/
[78] Solr official website, Accessed: Aug. 8, 2020. [Online]. Available: https://lucene.apache.org/solr/
[79] B. Fitzpatrick, “Distributed caching with memcached,” *Linux J.*, vol. 2004, no. 124, 2004, Art. no. 5.
[80] MySQL official website, [Online]. Available: https://www.mysql.com/
[81] CloudSuite 3.0 GitHub page, Accessed: Aug. 8, 2020. [Online]. Available: https://github.com/parsa-epfl/cloudsuite
[82] C. Delimitrou and C. Kozyrakis, “ibench: Quantifying interference for datacenter applications,” in *Proc. IEEE Int. Symp. Workload Characterization*, 2013, pp. 23–33.
[83] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[84] R. Pascanu, Ç. Gülçehre, K. Cho, and Y. Bengio, “How to construct deep recurrent neural networks,” *CoRR*, vol. abs/1312.6026, 2013. [Online]. Available: http://arxiv.org/abs/1312.6026
[85] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[86] Grafana official website, Accessed: Aug. 8, 2020. [Online]. Available: https://grafana.com/
[87] S. J. Pan and Q. Yang, “A survey on transfer learning,” *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 10, pp. 1345–1359, Oct. 2010.
[88] M. Caneill, N. De Palma , A. Ait-Bachir , B. Dine, R. Mokhtari, and Y. G. Cinar, “Online metrics prediction in monitoring systems,” in *Proc. IEEE Conf. Comput. Commun. Workshops*, 2018, pp. 226–231.



Dimosthenis Masouros (Member, IEEE) received the diploma degree in electrical and computer engineering from the National Technical University of Athens, Greece, in 2016. He is currently working toward the PhD degree at the National Technical University of Athens, Greece. His main research interests include resource management techniques for Cloud architectures. He has published more than 10 technical and research papers in international conferences and journals. He has also worked in EU H2020 projects AEGLE and EVOLVE.

Sotirios Xydis (Member, IEEE) received the PhD degree in electrical and computer engineering from the National Technical University of Athens (NTUA), Greece, in 2011. He is an assistant professor (tenure track) with the Department of Informatics and Telematics, Harokopio University of Athens. Then he worked for two years as post-doctoral researcher with the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) at Politecnico di Milano. He has worked for several years as a senior research associate of the Microprocessors and Digital Systems Laboratory at NTUA. His research interests include high performance computing, many-core and many-accelerator mapping and autotuning, design space exploration, high-level synthesis, accelerator design, reconfigurable processors, memory management, and power/thermal aware design for many-core platforms. He has published more than 100 papers in international journals and conferences, and he has received three best paper awards in the 2nd IEEE NASA/ESA Conference on Adaptive Hardware Systems (AHS 2007), in the 5th International Workshop of Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA 2013), and more recently in the 17th ACM International Conference on Computing Frontiers (CF’20). Also, he received HiPEAC Award for a DAC 19 paper for approximate computing. He was senior investigator in several EC funded programs and particularly focusing on scalable and high-performance Big Data infrastructures. He is a member of the Technical Chamber of Greece.

Dimitrios Soudris (Member, IEEE) received the diploma and PhD degrees in electrical engineering from the University of Patras, Patras, Greece, in 1987 and 1992, respectively. Since 1995, he has been a professor with the Department of Electrical and Computer Engineering, Democritus University of Thrace, Xanthi, Greece. He is currently a professor with the School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece. He has authored or coauthored more than 450 papers in international journals/conferences. He has coauthored/coedited seven Kluwer/Springer books. He is the leader and a principal investigator in research projects funded by the Greek Government and Industry, European Commission, ENIAC-JU, and European Space Agency. His current research interests include High-performance Computing, embedded systems, reconfigurable architectures, reliability, and low-power VLSI design. He was a recipient of the Award from INTEL and IBM for the EU project LPGD 25256 and the ASP-DAC 05 and VLSI 05 awards for EU AMDREL IST-2001-34379, as well as several HiPEAC awards. He has served as the general/program chair in several conferences.