2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD) | 978-1-6654-5155-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/SBAC-PAD55451.2022.00034

IntP: Quantifying cross-application interference via system-level instrumentation

Miguel G. Xavier, Carlos H. C. Cano, Vinícius Meyer, Cesar A. F. De Rose
School of Technology, *Pontifical Catholic University of Rio Grande do Sul (PUCRS)* - Porto Alegre, Brazil
miguel.xavier@pucrs.br, carlos.cano@edu.pucrs.br, vinicius.meyer@edu.pucrs.br, cesar.derose@pucrs.br

***Abstract*—Large-scale container datacenters host tens of thousands of diverse container-wrapped applications each day improving resource usage and maintenance costs. However, resource contention-related interference between co-located applications can severely degrade performance, affecting the quality of service at the user level and compromising experience. Understanding the sources of noise that generates this interference and better managing how to consolidate applications to physical hosts can significantly improve resource usage and overall performance reducing costs for providers and users. This paper presents IntP—an open-source system-level monitoring tool, which analyses selected architectural counters and operating systems data structures to estimate the stress an application puts on each hardware's subsystem and consequently infer the potential interference it could generate in other applications hosted in the same physical machine. Different from state-of-the-art tools that apply a more high-level approach using micro benchmarks and application metrics, IntPs low level instrumentation enables a more accurate prediction of the performance degradation that results from contention on shared resources, with less monitoring overhead. This information can be used to optimize scheduling strategies, which will make datacenter more resource-efficient and cost-effective. To show examples on how to use this tool and validate its results we present three cases studies that applied IntP in their interference-aware methodologies to improve resource utilization in distributed architectures that were able to achieve an increase up to 35% in resource efficiency and up to 25% in user level performance.**

***Index Terms*—Resource contention, Performance Interference, System-level Instrumentation, Monitoring**

I. Introduction

Large-scale container datacenters host tens of thousands of diverse container-wrapped applications each day improving resource usage and maintenance costs. However, interference between co-located applications – the overhead generated in an application running in a consolidated environment with other applications due to contention in accessing shared resources as CPU, memory, disk, and network – can severely degrade performance, affecting the quality of service at user level and compromising experience.

Preventing interference is a challenging task when managing large-scale heterogeneous data centers. Allocating one host for multiple applications increases cost efficiency and achieves better scalability, but the applications' performance tends to vary unpredictably, and the performance guarantee is likely compromised. Even allocating different applications to different processors' cores can induce the resources to a state of contention, as they share uncore caches (LLC), memory buses, storage subsystems, and the network layers [1], [2], [3].

The processor industry is trying to avoid contention with alternative cross-core networks and dedicated buses from processors to memory slots, promoting higher isolation across CPU's cores with lower interference. Even though it has opened new horizons for more controlled and isolated CPU architectures, contention still exists since the capacity of devices to handle interruptions is not superior to the capacity of buses to carry data, leading to bottlenecks that come from hardware to operating system drivers and application buffers. For example, multiple application tasks randomly accessing an HDD disk make its cylinder rotate asynchronously, reducing the number of segments it can write per unit. Or when cache sensitive tasks write and read data into the same shared cache, making their cached pages dirty and impossible to be evicted and reused. Many works have explored this topic in recent years, especially on cache contention. StatCC [4], for example, uses the StatStack [5] statistical model to efficiently estimate the stack distance [6] of an application to predict the cache miss rate for LRU caches. The cache miss ratio is commonly used to estimate performance (CPI) and model cache contention of co-scheduled applications. However, this also requires the cache pages to be referenced with task labels to estimate cache occupancy per task. Recent technologies have been embedded in processor chips to allow operating systems to collect cache occupancy per application task, allowing instrumentation with lower intrusion and performance overheads.

In this paper, we present IntP, a tool which analyses selected architectural counters and operating systems data structures from the host machine to estimate the stress running applications put on each hardware's subsystem and consequently infer the potential interference they could generate in other applications hosted in the same physical machines. Results provided by IntP can assist data center administrators to create scheduling strategies to place applications that stress the same hardware subsystems onto different machines thus reducing or even avoiding interference among them. We validate our tool presenting three case studies that exemplify how IntP was applied in their interference-aware strategies to improve resource utilization in distributed architectures that were able to achieve an increase up to 35% in resource efficiency and up to 25% in user level performance. Specifically, our contributions are as follows:

- We propose a methodology to quantify cross-application interference in consolidated environments based on low

2643-3001/22/$31.00 ©2022 IEEE
DOI 10.1109/SBAC-PAD55451.2022.00034

level instrumentation that enables a more accurate prediction of the performance degradation that could result from contention on shared resources without previous knowledge of applications and with very low monitoring overhead;
- We introduce an open source tool called IntP [1] that implements this methodology and can be easily incorporated to interference-aware strategies;
- We demonstrate with a experiment with BigData-centric applications how IntP can be leverage to improve interference-aware scheduling policies;
- We present two other case studies from related work showing how seamlessly IntP can be applied in different interference-aware scheduling scenarios and how resource utilization and performance was improved by its use.

II. Background

Uncontrolled access to shared resources can cause performance variations that lead applications to fail or run unsteadily. The friction generated by the competition to access RAM, disk storage, cache, or internal busses is called resource contention. Many efforts have been made to alleviate contention at the operating system level, ranging from better scheduling techniques in multicore architectures [7] to dynamically addressing mapping to minimize memory contention [8]. The steady growth of virtual data centers has raised a concern about resource contention and its impact in environments where performance is crucial, and SLA cannot be violated, such as clouds. For instance, I/O contention occurs when multiple tasks compete for a portion of disk bandwidth in a scenario where the demand is higher than the available resources. To illustrate, the dispersion in Figure 1 presents a contentious scenario in which two disk-intensive applications simultaneously write to/read from a single disk while the bandwidth is not sufficient to carry all data segments to disk without performance impacts.

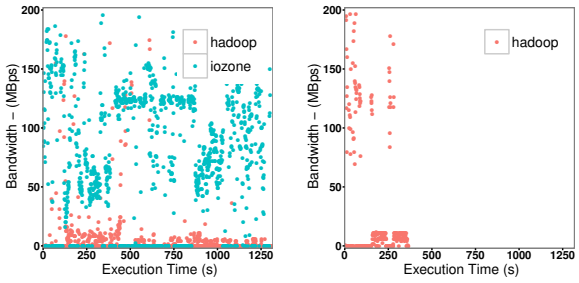

Fig. 1. Performance interference between two co-located applications due to the disk contention

On the other hand, performance interference may also arise due to isolation issues in the virtualization layer, which occurs when a virtual instance exceeds the number of allocated resources. Because resource limit settings are capacity-driven (e.g., GB, VCores, etc.) and not throughput-driven (e.g., bandwidth, IPC, etc.), even though a virtual instance receives a limited portion of the resource, there is nonetheless leakage due to uncontrolled access to operating system queues and uncore hardware components [9], [10]. Datacenter administrators have exaggerated the allocated resources to sidestep contentious scenarios, making the data center underutilized.

A. CPU Contention

Multiple CPU-intensive applications content for CPU when the tasks require a large number of cycles per unit of time to execute and the OS is barely able to allocate it for every instruction that is needed. This contention scenario is essentially observed in virtualization systems in which multiple virtual CPUs pinned over the same physical CPU are assigned to many instances running on the same computer. It makes virtual instances wait in a ready-to-run state before they can be scheduled on a CPU. The best defense against CPU contention is knowledge and understanding of the characteristics of the application and taking into that account while making better scheduling decisions, placing it together with applications that are the least likely to cause performance interference.

B. Memory Access Delay

Memory contention occurs when active tasks exceed the available physical memory. In a memory contention state, the system can not provide enough memory space for the tasks to run, and it starts to crash. Memory contention also prevents the CPU cores from achieving their peak performance. To address this problem, the OS starts to move fractions of active processes to the disk and tries to recover physical memory and reestablish stability. This management strategy is called system paging. Another alternative is to swap an entire process to the disk to reclaim memory, causing high disk overheads. This is an emergency technique used to combat extreme memory shortages, called system swapping. It is relatively difficult to avoid system paging and swapping, and in the end, there are only two simplistic possibilities to optimize memory performance: make more memory available for what the processes depend on most or decrease the extent of the competition for tasks. Unfortunately, if the users continue to spawn more tasks, the system will continue to induce performance overloads in memory, I/O, and consequently CPU [11].

C. Cache Pollution

LLC memory is a hardware device created to accelerate the speed of accessing data contained in RAM. It reduces the system bus and RAM traffic and restores recent translations from the virtual memory to the physical one. This procedure is also defined as the principle of locality. When two or more tasks are assigned to the same CPU node, tasks occasionally share on-chip memory space, which may lead to contention. This occurs when a greedy task pollutes LLC pages with data that is never reused, forcing other co-located cache sensitivity tasks to fetch data from RAM most of the time [44]. This is

[1] https://github.com/projectintp/intp

the case with streaming applications. On the other hand, when co-located tasks are cache-sensitive, the level of occupancy (capacity) should be taken into account during scheduling to minimize the cache miss ratio—the number of cache misses in the function of data loading.

D. Block Storage Latency

Disk throughput can be seen as the most volatile performance metric in a system because it is architecture-driven and might be affected by external components, such as virtual memory, bus, and I/O controllers. OS level I/O schedulers, such as CFQ, deadline, and noop, detect resource utilization bottlenecks and attempt to divide block devices by reordering/prioritizing operations in a fairly-balanced manner. As a result, the overhead is distributed equally across applications, but performance still varies unpredictably since the schedulers are unable to predict and make decisions based on workload characteristics. Applications might suffer from interference when there are consecutive random operations arriving in the disk. Then, the head assembly rotates to the track of the disk where the data will be read or written. This scenario makes the disk become busy while the I/O bus is kept below its full capacity. Furthermore, when short expressive operations (under 4KB) arrive in the disk, it makes the disk handle a bunch of operations without reaching its maximum throughput.

E. Network Back-pressure

Network card vendors have often changed the way that packets are handled from the hardware buffers up to the networking data path of the operating system. The faster the network devices become, the more processing time is necessary to handle hardware interrupts and process incoming packets at the same rate as they arrive. The time for processing a packet is strongly related to the multitude of protocol functions that it passes through after being fetched from NIC internal buffers and before reaching application sockets. In NUMA (Non-Uniform Memory Access) architectures, where there are different costs for accessing memory across CPU sockets, it becomes even worse. When data has to be traversed between the sockets, it consumes CPU cycles resulting in less work per unit of time since the tasks consume resources to deal with the cross-talk. A great deal of work has been done with the Linux kernel over the past few years, but the improvements sometimes depend on the workload type and are not always system-agnostic. Therefore, the system must be manually tuned to adjust depth queues, flow control, DMA delay, etc. With an understanding of the underlying factors that actually affect network packet processing and the need to do so, it is possible to minimize overheads and mitigate the network back-pressure.

III. IntP: Quantifying Cross-application Interference

In this section, we present the architecture and the built-in components of IntP, a tool that not only quantifies the interference application's tasks cause on hardware resources, but also provides insights about application demands during runtime. Since such an application has been instrumented, the users are capable of deciding which piece of hardware is more likely to be the bottleneck and making a decision about the queued application that best interleaves with it. Or, if one application starts to affect others, it could be migrated to other machines to minimize interference and maximize performance.

Some requirements were posed during the designing phase, including (1) IntP should not be intrusive to workloads; (2) IntP should run during application runtime; (3) IntP should instrument an application independent of how many applications are running together in the same machine. These requirements were raised considering limitations found in state-of-the-art solutions and are part of our effort to be constraints in this work. We developed IntP at the operating system level (kernel mode). This allowed us to instrument all OS components from drivers to scheduler queues with minimal performance intrusion. Figure 2 depicts IntP's components and the relationship between all of them.

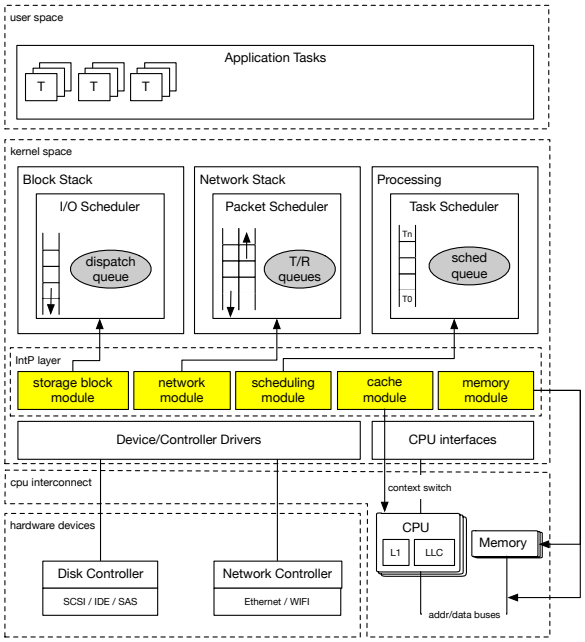

Fig. 2. Communication of IntP with kernel subsystems

Every time a user space task (userland) is waiting for an I/O event or synchronization operation to be completed, the OS needs to execute privilege instructions to take its place. This operation requires the OS's dispatcher to perform a context switch, saving the current task state in PCB to be restored/resumed later. The IntP's scheduling module is responsible for measuring the number of context switches a task performs per unit of time and provides the level of interference that a task can generate for other tasks during its lifetime. While in kernel mode, such a task can be waiting for storage block I/O interrupt, meaning that it invokes a disk-

related system call (read, write, seek, etc.) and is waiting for data to be retrieved from the disk. All data requests that come from userland tasks are queued, scheduled, and dispatched as the I/O controller is able to handle new requests. Hence, many queued requests make the I/O controller overloaded and unable to handle requests at the same rate as they arrived, given that the disk speed is slower than the CPU. The IntP's block storage module quantifies the level of pressure a task puts on the I/O dispatch queue, classifying those that are disk-intensive from those that do not disturb the I/O queue. The IntP's network module works similarly, but it relies on transmitted and received queues of network buffers, assessing network back-pressure generated per task. All instruction requires memory to be mapped and switched from user to kernel stack. However, there are tasks that require even more memory to process their instructions. This is the case for memory-intensive tasks such as those that belong to machine learning or streaming applications. These tasks not only use a lot of RAM memory to compute data but also pollute the CPU cache while running on it. The IntP's memory module connects to the CPU to collect per task cache occupation and derives with cache hits to generate cache sensitivity level. The level of memory bandwidth usage is also measured to classify memory-intensive tasks and differentiate them from cache-intensive tasks.

IV. System-level Resource Contentious Instrumentation

Unlike current solutions, IntP is composed of a set of modules running at the operating system level, which collect metrics from different hardware subsystems and operating system levels. Once started, the modules consist of hooks that probe operating system functions and apply filters on every instruction that comes from tasks to the hardware. For the case of storage block and network stack, interference may come from scheduling queues, and the dispatch rate is governed by the synchronism between the operating system and an external timer clock. This synchronism is architecture-dependent and comes from an external hardware timer that fires interrupts (jiffies) in time intervals of 1/HZ, where HZ is a compile-time constant that varies from 100 to 1000 in modern operating systems. Hence, the variables analyzed by IntP to assess interference in scheduling queues are defined as follows:

TABLE I
Queue Instrumentation Variables

Variable	Description
υ	average service time
γ	arrival rate
t	elapsed time
HZ	timer interrupt rate

The service time per unit of time is defined by:

$$f(t) = \frac{\upsilon * \gamma}{t} \tag{1}$$

Considering that the operating system performs scheduling decisions at intervals denoted by HZ, we divided the service time by HZ and integrate it from the instant t_0 to t_1:

$$I_{queue} = \int_{t_0}^{t_1} f(t)/HZDt \tag{2}$$

It means that each time the operating system looks at a scheduling queue, a job may or not be in progress. This assumption gives us the level of stress that an application is putting on queues over the operating system level at instant time t. The next subsections describe IntP instrumentation points that collect the above-mentioned variables and other interference perspectives that IntP is capable of inferring.

A. Block Layer Points

Despite many optimization techniques that have been developed, such as page caches for Writeback operations, the performance of block devices has a big impact on overall system performance. When a block request arrives into elevator scheduling queues, the scheduler does optimization functions (sorting, merging) in request queues to get efficient I/O. It means that requests are merged with others if either request ever grows large enough that they become contiguous. Afterward, they are sorted, not allowing a read to be moved ahead of a write or vice-versa. These optimization algorithms allow more contiguous read/write operations dispatched to disks, reducing seeks, and head movements in hard drives per unit of time. However, the higher the number of requests arriving at the elevator queues, the less efficient the general operation becomes since the disk handles incoming requests at lower rates than the CPU. This overload increases the queue depth (number of pending requests) and becomes even more noticeable in SMP machines, on which multiple tasks contend for a single disk.

A good metric to assess performance is defined by the time the disk takes to handle a request (i.e., service time). In order to infer the service time, we measured the delta-time from the *block_rq_complete* to *block_rq_issue* kernel functions. These points are called whenever a block segment is added and removed from the scheduling queue after the optimizations have taken place. Based on this, we measured the average service time υ (in milliseconds) for I/O requests and the arrival rate γ to quantify interference in the elevator queue. This interference metric is referred to as I_{disk} in the IntP.

B. Network Stack Points

With advances in CPU architectures and operating system structures, the network performance has also been improved in modern operating systems by changing packet receipt from interrupt-driven to polling mode. Previously, the network cards would typically fire a hardware interrupt whenever a packet arrives, causing the suspension of the executing software, and affecting application performance. Current operating systems have changed the way that network packets are handled once they are pulled off the wire. They implement a polling mechanism that is periodically interrupted. While the poll

method is executing, receive interrupts for the network device is disabled. The effect of this is that the operating system can drain potentially multiple packets from the network device receive buffer, increasing throughput, and decreasing latency at the same time as reducing the interrupt overhead. In an operating system based on Linux, the packet processing begins when the interrupt handling process (*ksoftirqd*) determines that a *softirq* is pending. It calls the *net_rx_action* driver-specific method, which begins processing all packets available in the network device ring-buffer before its CPU time is up (limited to 2 jiffies). The processing ends up when the data is copied to an application-specific socket buffer. It turns out that at this point, applications still suffer from throughput issues due to back-pressure caused by cross-application tasks, making either the interrupt handling mechanism unable to drain packets from the network device fast enough or the application unable to dequeue packets from the socket buffer fast enough.

We focused on analyzing the network packet path from the network device (ring buffer) to the application buffer (socket's receive buffer) or vice-versa so that an application can be classified by its level of pressure placed on the hardware device (throughput) and operating system's network stack (latency). The latency is meant as the average service time υ. Since the OS's network stack controls two-ways communications (send/recv) using different queues, the IntP should instrument the scheduler functions in isolation. The average service time of the sending queue is obtained by the delta-time from the *net_dev_xmit* to *__dev_queue_xmit* functions. And the average service time of the receiving queue is obtained by the delta-time from the *napi_complete_done* to *__napi_schedule_irqoff* functions. The average service time υ is given by the sum of both metrics. The arrival rate γ is given by the total of send and receive packets per unit of time. This interference metric is referred to as $I_{netstack}$ in the IntP.

On the other hand, IntP aims to measure the interference sourced from contention in the network card, which occurs when the bandwidth is not enough for multiple tasks to carry all the data that is needed (i.e., capacity overflow). The bandwidth consumed per task is obtained using the probes as above, but accumulating the length of each packet dispatched and received per unit of time. Hence, the interference from the hardware device is given by:

$$I_{netcapacity} = \int_{t_0}^{t_1} \frac{SUM(length)}{bandwidth} \tag{3}$$

Where bandwidth is the nominal limit of the network card capacity.

C. CPU and OS Dispatcher Points

The IntP's scheduling module collects the context-switch(CSW) metric. When a context-switch occurs, the module probes the OS's dispatcher process and accumulates the event-waiting time α for each application's thread in the blocking state waiting for I/O or system call. IntP ignores the waiting time in preemptive operating systems when quantum expires. The delta-time gives the waiting time of a thread in the blocking state between the instant that it was preempted and resumed back to the CPU. The waiting time is collected for all context-switch operations throughout the task runtime in intervals denoted by t_0 and t_1 as follows:

$$\alpha_{th} = \int_{t_0}^{t_1} CSW_{time} \Delta t \tag{4}$$

Given that an application may be multithreading, then we need a discrete equation to sum the waiting time α of a set of threads. Thus, let $S : S \subseteq E$ be a subset of threads running in the system E. The context-switch instrumentation metric of the application's threads S is given by

$$I_{csw} = \sum \alpha_{th}, \forall_{th} \in S \tag{5}$$

D. Memory Points

IntP aims to assess the level of interference an application causes during memory access. The IntP's memory module collects counters from the memory controller, a digital circuit that manages the flow of data going to and from the main memory. It is usually called Integrated Memory Controller (IMC). The first approach was to use LLC_MISS (last level cache miss) * 64 Bytes (size of a cache line). However, the problem with this approach is that the LLC_MISS counter would not include prefetch misses. This can be a huge issue when there are a lot of prefetching activities involved (for example, when there is streaming access involved in the program). Recent CPU architectures made available counters that can be fetched from the uncore IMC, allowing more precise observations. Hence, the level of interference an application puts on memory access is given by:

$$\gamma_{th} = \int_{t_0}^{t_1} (MRC + MWC) * CLDt \tag{6}$$

Where MRC and MWC denote the number of reads and memory writes, respectively. And CL is the size of cache line (commonly 64). Finally, the integration of the application's threads is summed as follows:

$$I_{mem} = \sum \gamma_{th}, \forall_{th} \in S \tag{7}$$

By normalizing I_{mem}, IntP outputs a metric (0..1), which ranges from lowest to highest interference degree, of which is possible to infer the behavior of the application's threads while they are accessing the main memory.

E. LLC Points

The last level cache is a key resource to manage since multi-threaded architectures and multicore platforms are constantly arising. The chip industry has been introducing a new feature in the hardware that allows an OS to determine the usage of the cache by applications running on the platform. This is the case with Intel Cache Monitoring Technology (CMT) [12]. CMT provides mechanisms for an OS to indicate a software-defined ID for each of the threads that are scheduled to run on a core. This ID is called the Resource Monitoring ID (RMID).

Since there are associations between threads and RMIDs, they are programmed via a thread-specific model-specific register called MSR and can be read by system software at any time through an MSR interface. The built-in cache module of IntP takes advantage of this feature and begins mapping the application's threads to RMIDs during runtime to infer per-application cache usage; thus, cache interference can be denoted by;

$$\theta_{th} = \int_{t_0}^{t_1} MSR(rmid_{th})Dt \tag{8}$$

Where MSR is the interface that read the thread-specific $rmid$ from the CPU register during the instant time t. Finally, the total cache occupancy of an application is given by:

$$I_{cache} = \sum \theta_{th}, \forall_{th} \in S \tag{9}$$

IntP monitors an application process that it expects as a parameter. It profiles the application during runtime, returning the interference the application generates on each subsystem. Moreover, IntP returns the interference metrics, in percentage, normalized, where the higher the metric is, the more interference the application being profiled generates. IntP differs from other resource usage tools since it inspects OS's internal components to infer contention-related performance interference due to bottlenecks in I/O queues, buffers, and uncore buses. From the memory's point of view, an application that allocates 80% of memory would not imply that it is stressing the memory, as it could just have it allocated and not do further operations. On the other hand, an application that is using only 20% could be doing a great amount of reading and writing operations to the memory; thus, it would generate a higher interference. These interference levels, though, are measured by IntP.

IntP outputs interference metrics for CPU, disk, memory, network, and cache. More specifically, it returns the following metrics:

- **netp** - percentage of physical network interference
- **nets** - percentage of network queue interference
- **blk** - percentage of disk interference
- **mbw** - percentage of memory bus interference
- **llocc** - percentage of cache interference
- **cpu** - percentage of cpu interference

Figure 3 shows the interference levels generated by a given application while varying its workload. This application is disk-intensive and has a high network affinity with another application. It is noticeable that the interference levels tend to increase as the workload also grows. Moreover, the contention in disk storage has a unique behavior, which varies between every data collection. This behavior is due to the write operations that are being executed in an asynchronous manner.

V. IntP-assisted case studies

In this section we validate our tool presenting three case studies that exemplify how IntP was applied in their interference-aware strategies to improve resource utilization and user level performance in distributed architectures. Important to note that [13] showed that IntP profiling was non-intrusive, with very low overhead even with hundreds of running applications.

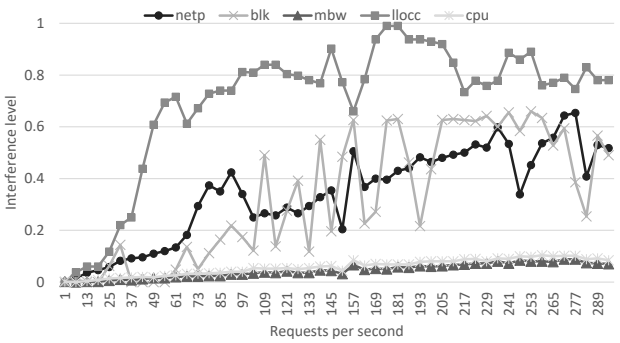

Fig. 3. Example of an IntP output for a disk-intensive application.

A. BigData interference-aware Task Scheduling

In this first case study we demonstrate how IntP can be leveraged to improve interference-aware scheduling policies, in this case for better BigData-centric application scheduling. IntP was used to assess interference metrics of heterogeneous applications that put stress on different hardware components and OS subsystems. We selected popular benchmarks from HiBench Benchmark Suite [14], which are well-known representatives in the field of data analytics. The applications were chosen and classified by their resource intensity levels, such as cache intensive, compute-intensive, and disk-/network-intensive. Such classification covers contention scenarios that IntP proposes to the instrument. The applications we choose are presented in Table II.

TABLE II
WORKLOAD CHARACTERISTICS

App	Type	workload
App01	machine learning	LLC
App02	machine learning	LLC
App03	machine learning	LLC
App04	streaming	LLC/memory
App05	streaming	LLC/memory
App06	ordering	memory
App07	ordering	memory
App08	classification	CPU/memory
App09	classification	CPU/memory
App10	search engine	CPU
App11	sort	network
App12	sort	network
App13	query/scan	disk
App14	query/join	disk
App15	query/merge	disk

Our hardware setup comprises 16 identical Dell PowerEdge R810 machines. Each of them is equipped with two 3.46Ghz Intel Xeon C5690 processors with eight cores each (with Hyper-Threading), totaling 32 virtual cores; 64Gb of RAM memory, and four Gigabit Ethernet adapters. The communication between them is done via a Gigabit switch. We deployed

the Linux distribution Ubuntu 16.04 onto the machine. The IntP was compiled and loaded into the kernel with all modules enabled. The applications were scheduled on the experimental testbed in a 1-after-1 manner to collect the IntP metrics for each application individually. Figure 4 presents the interference ratios obtained from IntP for each application.

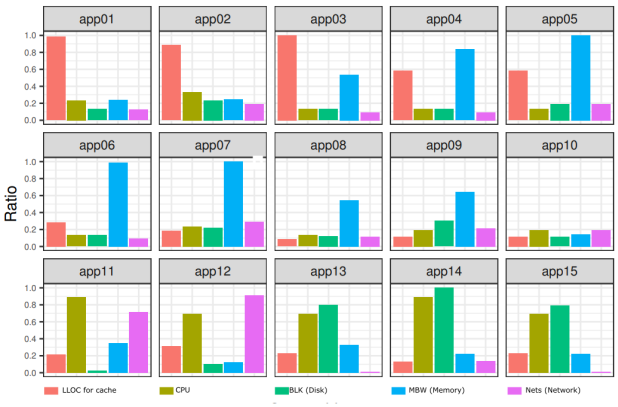

Fig. 4. IntP instrumentation outcome per application (llocc for cache in red, cpu in brown, blk for disk in green, mbw for memory in blue and nets for network in pink)

We can obtain some insights from the similarity among the applications. It is easy to see that by placing applications that least interfere with each other on different computing nodes, it would be possible to minimize resource contention and maximize performance. To classify this level of similarity among them, we need to reduce the dimensions from 5D to 2D. Hence, we used the statistical procedure called Principal component analysis (PCA) [15], which uses an orthogonal transformation to convert a set of observations/results or correlated variables into a set of values of linearly uncorrelated variables, also called principal components. Generally, PCA results are less than or equal to the number of original variables. As a result, we generated a 2D representation that allowed us to see the interference-related proximity between applications. The PCA results lead us to an optimization problem (clustering analysis) in such a way that we can find the K cluster centers and assign the objects to the nearest cluster center, where the squared distances from the cluster are minimized. We applied a centroid-based clustering analysis using the *K-means* method to group variables (i.e., applications) by their similarity. As we are interested in using IntP results for better scheduling and placement strategies in computer clusters, the value of K may be defined considering the number of cluster nodes. The higher the number of nodes, the greater the granularity of K to accommodate applications with the least possible interference among them we excluded $K = 3$ because one cluster would have 10 applications (red+blue) and the model would try to avoid putting them together in the same physical node. The clustering results for $K = 4$ is presented in Figure 5. Results produced using cluster analysis can now be used to assist system administrators during a placement process, in which applications that least interfere with one another are consolidated.

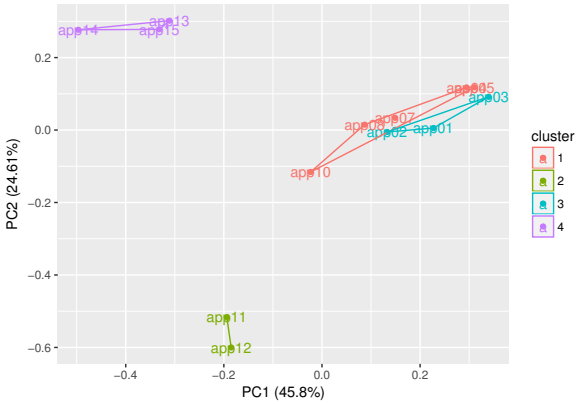

Fig. 5. K-means with K=4

From our analysis, we implemented an interference-aware task scheduling in Apache Hadoop YARN [16]. YARN is the architectural center of Hadoop that allows multiple data processing engines such as interactive SQL, real-time streaming, data science, and batch processing to handle data, as such the applications we have used during our analysis. We selected a set of applications from different frameworks and programming engines to extend heterogeneity, including Hadoop, Spark, and Storm. In addition, we chose the YARN's Fair policy (default installed) to compare it with the proposed interference policy. We used a carefully-crafted external script to connect to the YARN's client API and work like the dispatcher moving jobs every 5 seconds on the 10-in-10 order (no job completion waiting). The experiment aims to evaluate the jobs' turnaround times (makespan) and total completion times. The performance evaluation, as well as the comparison with the default YARN scheduler, is presented in Figure 6.

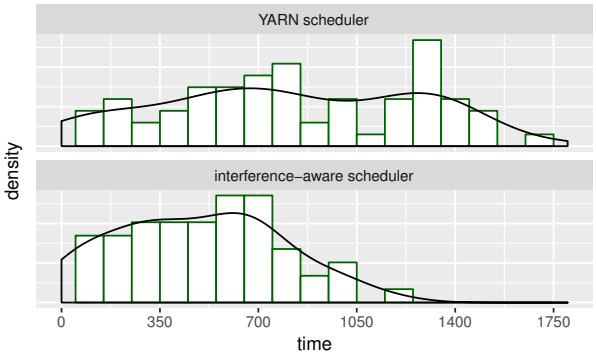

Fig. 6. Comparison between intp-based scheduler and YARN's scheduler. Density represents the number of jobs completed per time slice.

The graph shows that the reduced job turnaround times reflected in the total completion time, and also improved the

efficiency (density) expected when evaluating performance in scheduling. We observed a performance optimization of up to 35%. This is because applications have been better balanced according to their interference level so that they compete less for resources.

B. Interference-aware multi-tier application placement

Cloud providers are constantly seeking to become more cost effective, where a common strategy is to consolidate multiple applications in physical machines using virtualization techniques. This consolidation, however, may result in performance related problems such as resource interference. Moreover, if the workload is composed of multi-tier applications, an increasingly popular method of application development, especially for web and mobile, in which tiers need to communicate through the network, we have another possible source of performance degradation, which we refer as network affinity. In order to reduce the effects of such problems, Ludwig et al. [17] explored the problem of multi-tier application placement in consolidated environments, focusing on resource interference and network affinity. They propose placement policies and algorithms and evaluated them for different workload scenarios using a simulation tool we developed called CIAPA (Capacity, Interference and Affinity-aware Placement Algorithms). CIAPA introduced a performance degradation model, a cost function, and heuristics to find an optimized placement for a specific workload of multi-tier applications.

CIAPA relied on IntP to individually profile all parts of each application of a specific workload, gathering their interference levels, so that each of them is labeled and treated as a separated scheduling unit. This is possible because IntP is able to measure these metrics for specific processes, based on a process ID (pid). All this information is then passed to the performance degradation model, and a cost function and heuristics will be used to decide the best placement. This strategy was validated with four scenarios, being of them a set of 50 tiers with a mixed distribution of used resources. This configuration is used to reproduce the same level of complexity we are seeing in real industry problems and also in related work. Tiers were divided into five groups, where each group used the following resources intensively: CPU, disk I/O, memory, cache, and mixed. Moreover, three tiers of each group had network affinity with other tiers in the same group. Other two tiers also had network affinity, but with tiers from other groups. This case aims at simulating a more controlled environment, where there is an equal distribution of utilized resources. Figure 7 shows the cost comparison for placements generated by the above strategies for the described scenario. It is noticeable that CIAPA using the best results by using Simulated Annealing (CIAPA-SA) generates placement configurations with much lower cost when compared to both strategies from related work (up to 60% reduction when compared to Affinity and up to 10% to Interference). This is expected since interference has a higher impact on the cost function and Affinity strategies do not optimize for it. Also noticeable is CIAPA-SA significant advantage in configurations with fewer PMs.

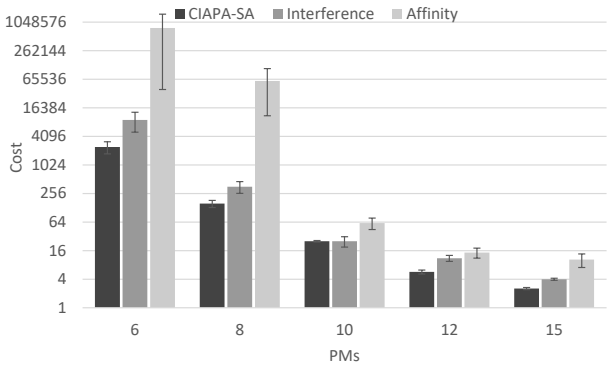

Fig. 7. Cost comparison of CIAPA-SA against related work by Physical Machines(PM). Adapted from [17].

Experimental results compared the solution generated by CIAPA-SA to other placement strategies from related work, and for the tested scenarios, it delivered placement decisions with better cost and, consequently, improved performance. They observed a reduction in response time of up to 10% when compared to interference strategies and up to 18% when considering only affinity strategies.

C. Dynamic interference-aware scheduling for latency-sensitive workloads

Performance interference among web applications is known to adversely impact Quality of Service (QoS) properties and Service Level Agreements (SLAs) of applications. Dynamic service demands and irregular workload profiles further raise the challenges for cloud service providers in managing resources on-demand to satisfy SLAs while minimizing operational costs [18]. To deal with these issues, Meyer et al. [13] developed IADA, a full-fledged dynamic interference-aware cloud scheduling architecture for latency-sensitive workloads. Motivated by experimental results that show that applications may change their interference profiles due to workload variations, their approach constantly reevaluates this interference metrics during execution to reduce cross-application interference and dynamically trigger rescheduling operations to improve resource usage and reduce SLA violations. To evaluate the proposed architecture, the authors utilized real workload traces, initially using a real cluster and subsequently scaling it out through simulation tests. In all experiments, IADA improved the average response time by 25% when compared to other scheduling approaches in similar scenarios.

To be able to accomplish this, IADA has a profiling phase that uses IntP to extract information on how individual applications stress hardware resources and, based on this, triggers rescheduling operations to minimize this overhead in nodes that are consolidating applications. To give a visual example of IntPs pivotal role in the proposed interference-aware architecture, Figure 8 shows how interference metrics are used to reduce overhead across consolidated applications. The red dashed line demonstrates the exact moment the scheduling

was executed. By looking at the interference measurements and the respective applications consolidated in each node, it is possible to see two consequences of this rearrangement: (i) app2 (disk intensive) and app6 (cpu intensive) applications (emphasized in bold) switched nodes; and (ii) because of that average interference levels are reduced in both nodes due to less cross-application interference.

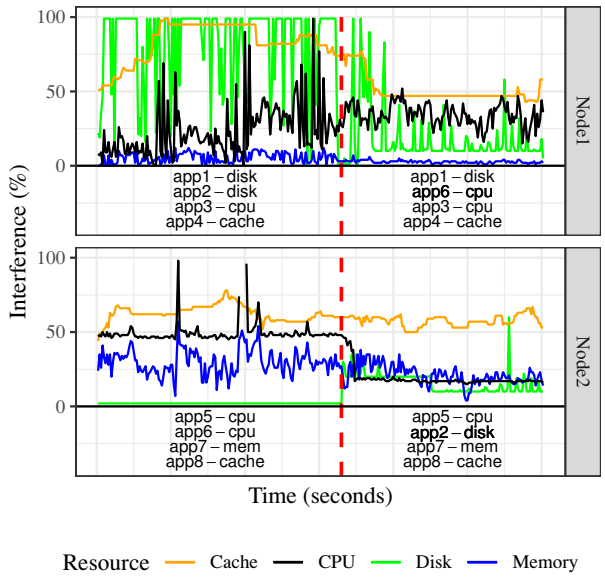

Fig. 8. IntP measured interference levels in two nodes while performing a scheduling operation (cpu in black, blk for disk in green, llloc for chache in orange, and mbw for memory bus in blue). Applications which have migrated across nodes after the rescheduling (red dotted line) are emphasized in bold. Adapted from [13].

VI. Related Work

Minimizing the performance interference effect from co-located applications is a trending topic nowadays, and it occurs due to the inherent nature of data-intensive distributed analytic frameworks to move large volumes of data to be processed in virtual data centers [19]. Recent research proposes quantifying performance interference in several virtualized scenarios [20] [21] [3] [22] [23].What we propose is a very accurate tool to quantify the performance interference for any workload by using selected architectural counters and OS data structures. Matthews et al. [24] present the Isolation Benchmark Suite (IBS), but it relies on deprecated micro-benchmarks that render it a very inaccurate and they have to be executed manually and individually, leading to error-prone, inaccurate, and unreliable results.

Tracon [25] uses machine learning algorithms for modeling performance interference on cloud environments. Then Eucalyptus [23] considers CPU load as the main factor instead of using other inputs as this tool provides focusing it on Web applications. Both tools are focused in Cloud and hypervisor environments while our tool is able to work in Bare metal, hypervisor or cloud and provide better information for interference profiling.

The benchmark proposed by Delimitrou et al. Ibench [26] presents highly reliable results, but it is limited to quantifying interference across multi-tenant workloads in traditional data centers, consisting of a set of carefully-crafted benchmarks that induce interference of increasing intensity in resources that span the CPU, cache hierarchy, memory, storage, and networking subsystems.Then neither the compiled IBench's program nor the source codes are available to the community. Our tool does not uses benchmarks nor increase load while running and is open source and available to anyone.

Bubble-Up [27] is a characterization methodology that enables the accurate prediction of the performance degradation that results from contention for shared resources in the memory subsystem. Its main focus the memory subsystem while our tool can provide interference metrics for IO, CPU, Disks, Network and memory.

Paragon [28] is an online and scalable DC (Data Center) scheduler that is heterogeneous and interference-aware. Paragon is derived from robust analytical methods and instead of profiling each application in detail, it leverages information the system already has about applications it has previously seen. It uses collaborative filtering techniques to quickly and accurately classify an unknown, incoming workload with respect to heterogeneity and interference in multiple shared resources, by identifying similarities to previously scheduled applications. The classification allows Paragon to greedily schedule applications in a manner that minimizes interference and maximizes server utilization.

Quasar [29] determines the right amount of resources to meet these constraints at any point. Second, Quasar uses classification techniques to quickly and accurately determine the impact of the number of resources (scale-out and scale-up), type of resources, and interference on performance for each workload and data set.

Recent works, such as Mage [30], have a more active role in using micro benchmarks and staged processes to infer the interference and using offline and online data to infer its interference.

Aforementioned tools above have an active role as using benchmarks to measure and profile [28] [2] [29] performance and how the applications will perform and [30] will use staged process to scale the load. Meanwhile the tool proposed is very passive and does not create additional load on the host system, allowing it be run anytime without any degradation. Hardware monitoring tools as Perfmon2 [31] provides an common interface to access CPU and hardware counters thru several architectures, its model focus onto providing an interface where you can create contexts to monitor the whole system or specific threads. That monitoring interface can have as many as 290 [32] raw specific counters to choose and select for. The tool described here is able to gather the needed counters and use it to report the interference the

application has on the host without any interruption or load. In addition to these techniques, other remote works have explored interference detection in hypervisor-based systems by collecting hypervisor's performance counters from its scheduling components. In contrast, but designed differently, IntP works within the OS's kernel like a minimalist, but non-intrusive resource sensitiveness reporting module.

VII. Conclusion and Future Directions

In this paper, we proposed a methodology to quantify cross-application interference in consolidated environments based on low level instrumentation. It enables a more accurate prediction of the performance degradation that could result from contention on shared resources without previous knowledge of applications and with very low monitoring overhead. Further, we introduced an open-source tool called IntP that implements this methodology and can be easily and efficiently incorporated to interference-aware strategies. To validate our methodology, we presented three cases studies where IntP was used in different distributed scenarios to quantify cross-application interference and how it contributed to improvements up to 35% in resource efficiency and up to 25% in user level performance.

In future work, we will improve IntP to be more compatible with newer hardware architectures and operating system kernels and also its support for virtualization environments implemented by either virtual machines or containers.

References

[1] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 22.

[2] 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 *44th IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 248–259.

[3] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[4] D. Eklov, D. Black-Schaffer, and E. Hagersten, "Statcc: a statistical cache contention model," in *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*. IEEE, 2010, pp. 551–552.

[5] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of lru caches," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 55–65.

[6] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[7] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM SIGARCH Comp. Arch. News*, vol. 38. ACM, 2010, pp. 129–142.

[8] K. H. Potter, "Dynamic addressing mapping to eliminate memory resource contention in a symmetric multiprocessor system," Jan. 7 2003, uS Patent 6,505,269.

[9] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[10] M. G. Xavier, M. V. Neves, and C. A. F. De Rose, "A performance comparison of container-based virtualization systems for mapreduce clusters," in *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 2014, pp. 299–306.

[11] M. Loukides and M. Loukides, *System Performance Tuning*, ser. Computer Science Series. O'Reilly, 1992. [Online]. Available: https://books.google.com.br/books?id=3qpQAAAAMAAJ

[12] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache qos: From concept to reality in the intel® xeon® processor e5-2600 v3 product family," in *IEEE Int. Symp. on High Performance Computer Architecture (HPCA)*, 2016, pp. 657–668.

[13] V. Meyer, M. L. da Silva, D. F. Kirchoff, and C. A. De Rose, "Iada: A dynamic interference-aware cloud scheduling architecture for latency-sensitive workloads," *Journal of Systems and Software*, vol. 194, p. 111491, 2022.

[14] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *New Front. in Inf. and Soft. as Services*. Springer, 2011, pp. 209–228.

[15] M. Mohammed, M. B. Khan, and E. B. M. Bashier, *Machine Learning: Algorithms and Applications*. CRC Press, 2016.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[17] U. L. Ludwig, M. G. Xavier, D. F. Kirchoff, I. B. Cezar, and C. A. F. De Rose, "Optimizing multi-tier application performance with interference and affinity-aware placement algorithms," *Concurrency and Computation: Pract. and Exp.*, vol. 31, no. 18, p. e5098, Sep 2019.

[18] V. Meyer, D. F. Kirchoff, M. L. Da Silva, and C. A. De Rose, "Ml-driven classification scheme for dynamic interference-aware resource scheduling in cloud infrastructures," *Journal of Systems Architecture*, vol. 116, p. 102064, Feb 2021.

[19] M. V. Neves, C. A. F. D. Rose, K. Katrinis, and H. Franke, "Pythia: Faster big data in motion through predictive software-defined network optimization at runtime," in *IEEE 28th International Parallel and Distributed Processing Symposium, IPDPS*, 2014, pp. 82–90.

[20] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," in *Proceedings of the 8th Int. ACM SIGSOFT Conf. on Quality of Soft. Architectures*, ser. QoSA '12. New York, NY, USA: ACM, 2012, pp. 91–100.

[21] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 51–58.

[22] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling virtual machine performance: Challenges and approaches," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 3, pp. 55–60, Jan. 2010.

[23] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[24] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, "Quantifying the performance isolation properties of virtualization systems," in *Proceedings of the 2007 workshop on Experimental computer science*. ACM, 2007, p. 6.

[25] R. C. Chiang and H. H. Huang, "Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2011, p. 47.

[26] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 23–33.

[27] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity," *IEEE Computer Architecture Letters*, vol. 10, no. 2, pp. 29–32, 2011.

[28] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[29] ——, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGPLAN Notices*, vol. 49, no. 4. ACM, 2014, pp. 127–144.

[30] F. Romero and C. Delimitrou, "Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems," in *27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18, New York, NY, USA.

[31] S. Eranian, "Perfmon2: a flexible performance monitoring interface for linux," in *Proc. of the 2006 Ottawa Linux Symposium*. Citeseer, pp. 269–288.

[32] Intel. Perfmon events. [Online]. Available: https://perfmon-events.intel.com