Pliant: Leveraging Approximation to Improve Datacenter Resource Efficiency

Neeraj Kulkarni, Feng Qi, and Christina Delimitrou
Cornell University
{*nsk49, fq26, delimitrou*}*@cornell.edu*

***Abstract*—Cloud multi-tenancy is typically constrained to a single interactive service colocated with one or more batch, low-priority services, whose performance can be sacrificed when deemed necessary. Approximate computing applications offer the opportunity to enable tighter colocation among multiple applications whose performance is important. We present Pliant, a lightweight cloud runtime that leverages the ability of approximate computing applications to tolerate some loss in their output quality to boost the utilization of shared servers. During periods of high resource contention, Pliant employs incremental and interference-aware approximation to reduce contention in shared resources, and prevent QoS violations for co-scheduled interactive, latency-critical services. We evaluate Pliant across different interactive and approximate computing applications, and show that it preserves QoS for all co-scheduled workloads, while incurring a 2.1% loss in output quality, on average.**

I. Introduction

Cloud computing has reached proliferation by offering *resource flexibility* and *cost efficiency* [1]–[4]. Resource flexibility is achieved by users elastically scaling their resources on-demand, and releasing them when they no longer need them. Cost efficiency is achieved through multi-tenancy, i.e., by co-scheduling multiple jobs on the same physical platform to increase server utilization. Unfortunately multi-tenancy also leads to unpredictable performance, due to interference in shared resources [5]–[20]. When the applications that suffer from interference are high priority, interactive services, such as websearch and social networking, multi-tenancy is disallowed altogether, hurting utilization, or - at best - interactive services are co-scheduled with low priority, best-effort workloads [9], [20], [21]. The performance of these workloads can be sacrificed at runtime to avoid performance degradation for the high priority service [7]–[9], [21]–[25]. Unfortunately this limits the options cloud providers have in terms of applications they can co-schedule with interactive, latency-critical services. Approximate computing offers the potential to break this utilization versus performance trade-off in shared clouds.

Approximate computing applications include workloads from several fields, such as computer vision, machine learning, analytics, and scientific applications, and have the common feature that they can tolerate some loss in output accuracy in return for improved performance and/or energy efficiency [26]–[30]. Several cloud workloads fall under this category, such as big data analytics and ML applications, where achieving the highest output quality is often less important than latency and/or throughput. Exposing the knob of approximation to the cloud scheduler allows the system to sacrifice some accuracy in applications that can tolerate it, to preserve the services' quality-of-service (QoS) constraints.

We present Pliant, an online cloud runtime system that achieves both high QoS and high utilization by leveraging the ability of approximate computing applications to tolerate some loss in their output quality. Pliant enables aggressive co-scheduling of interactive, latency-critical services with approximate applications. Unlike prior cluster schedulers, Pliant does not consider applications co-scheduled with interactive services as low-priority, and preserves their nominal performance [9], [18], [21]. Instead, a user expresses an application's tolerable inaccuracy threshold to the scheduler, and Pliant dynamically adjusts approximation to the minimum amount needed to satisfy the tail latency QoS of the interactive service over time, without exceeding this threshold.

Pliant consists of a lightweight performance monitor, an online dynamic compilation system, and a runtime resource controller. The monitor uses adaptive sampling of end-to-end latency to continuously check for QoS violations in the interactive service, while the compilation system is based on DynamoRIO [31], and adjusts the approximation degree of an application online to reduce interference in shared resources. When approximation alone is not enough to counter the performance impact of interference, the resource controller additionally reclaims resources from the approximate application(s), yields them to the interactive service, and adjusts the approximation degree to ensure that the execution time of the approximate application(s) does not degrade. Pliant reclaims resources incrementally to guarantee that the approximate application only sacrifices the minimum amount of accuracy needed at each point in time, and selects the type of resource to be reclaimed based on the utilization of different subsystems. Finally Pliant leverages a set of lightweight Linux signals to switch between approximation degrees, and only uses DynamoRIO at coarse - function - granularity, to avoid the high overheads of dynamic compilation.

We evaluate Pliant on servers with 44 physical cores, with three popular open-source interactive services; `memcached`, a distributed in-memory cache [32], `NGINX`, a front-end web server [33], and `MongoDB`, a stateful persistent database [34]. We additionally use a set of 24 scientific applications from PARSEC [35], SPLASH2 [36], BioPerf [37], and Minebench [38] as the approximate applications. We show that Pliant is able to preserve both the tail latency QoS of the interactive services,

and the nominal execution time of the approximate applications, with a 2.1% loss of output quality on average, and 5% loss in the worst case. In comparison, running the applications in precise mode results in a $2-10x$ increase in the tail latency of the interactive services, a dramatic degradation for latency-sensitive, interactive services. Finally, we explore the sensitivity of Pliant to input load, decision granularity, and worst-case inaccuracy threshold for each interactive service.

II. Related Work

We now review relevant work in interference-aware scheduling, approximate computing, and dynamic instrumentation.

Contention-aware scheduling: Sharing resources to increase utilization results in performance degradation [5], [6], [8], [13], and in some cases security vulnerabilities [11], [39]–[42]. Several systems that aim to minimize destructive interference disallow colocation of jobs that contend in the same resources [5]–[8], [13], [43], or partition resources to improve isolation [9], [11], [24], [25]. For example, BubbleFlux determines how the memory sensitivity of applications evolves over time, and prevents multiple memory-intensive services from sharing the same platform [6]. Similarly, DeepDive identifies the interference colocated VMs experience, and manages it transparently to the user [13]. Paragon [5] and Quasar [8] are cluster managers that leverage a set of practical online data mining techniques to determine the resource requirements of incoming cloud applications, and schedule them in a way that minimizes resource contention. In the same spirit, Nathuji et al. [7] develop Q-Clouds, a QoS-aware control framework that dynamically adjusts resource allocations to mitigate interference in virtualized clouds.

On the isolation front, Lo et al. [9] study the sensitivity of Google applications to different sources of interference, and combine hardware and software isolation techniques to preserve the QoS of latency-critical applications running alongside batch, low-priority workloads. Similarly, Kasture et al. [24] implement fine-grained cache partitioning, and power allocation with RAPL [25] on servers that host one interactive, and one or more best-effort services. In all cases, a server hosts at most one high priority application; any remaining workloads are best-effort, and their performance can be sacrificed when needed.

Approximate computing techniques: Finding the approximation potential of popular application classes, and generating language constructs to express and verify approximation has generated a large amount of related work. Carbin et al. [30] present language constructs for specifying acceptability properties in approximate programs, and develop a system that enables developers to obtain fully machine-checked verifications of their approximate applications. Sampson et al. [44] propose annotating data types that can be approximated, and automatically mapping such variables to low-power storage, and using low-power operations on them. They extend this work to map such variables to approximate storage devices in [26]. The same authors develop ACCEPT [45], a programmer-guided compiler framework that identifies approximable code, and automatically chooses the best approximation strategies for it. Finally, Misailovic et.al [28] present Chisel, an optimization framework that automatically generates approximate instructions and data that can be stored in approximate memory to improve energy efficiency, at the cost of some reliability and accuracy loss. They also propose compiler-level, accuracy-aware transformations that automatically generate approximate versions of programs [29]. A lot of this prior work focuses on improving the programmability and ease of development of approximate applications, to avoid tasking the user with generating approximate variants manually [28], [29], [44].

Dynamic recompilation: Open-source tools such as DynamoRIO [31] enable online code transformations that can be used, among other reasons, to reduce the amount of resource contention the instrumented application incurs in a multi-tenant system. For example, Protean Code [22] is a co-designed compiler and runtime built on top of LLVM that enables compiler transformations at runtime with less than 1% overhead. The runtime dynamically mitigates cache pressure via fine-grained code transformations that disable prefetching for the low-priority application during periods of high resource contention. There are also several dynamic optimization systems that do not directly aim to reduce resource contention, but focus on code transformations that optimize application performance. Mojo [46] was the first tool to facilitate dynamic software optimizations on an x86 architecture, while ADAPT [47] is a compiler-supported, high-level, adaptive optimization system, which leverages user-provided optimizations and heuristics to efficiently explore the application design space at runtime. Finally, ADORE [48], [49] is a dynamic optimization runtime that monitors performance counters during application execution to detect hotspots, and leverages online compilation using a system similar to DynamoRIO, to tune data cache prefetching. Trident [50] builds on ADORE, and uses hardware support to reduce the overhead of online application profiling.

III. Approximation Design Space Exploration

We first examine the potential that approximation offers in trading off quality for performance and efficiency in multi-tenant cloud settings. We explore several approximation strategies whose performance and efficiency benefits have been previously shown [28], [44], including *loop perforation*, *synchronization elision*, *lower precision data types*, and *tiling*.

- **Loop perforation:** This technique omits a fraction of the iterations of a loop. Typical approximate computing applications, like analytics and machine learning workloads are iterative in nature, making loop perforation a good candidate for approximation. There are multiple ways to perforate a loop. For example, to reduce a loop by a factor p, we can execute only a chunk of (MAX_ITER/p) iterations, or execute every p^{th} iteration. We can also reduce the loop by a factor of $(p-1)/p$ by not executing every p^{th} iteration. Perforating a loop lowers the accuracy of the output, however at the same time it also leads to lower execution time, and reduced memory traffic by avoiding the data accesses of the omitted iterations. Note that the decrease in output

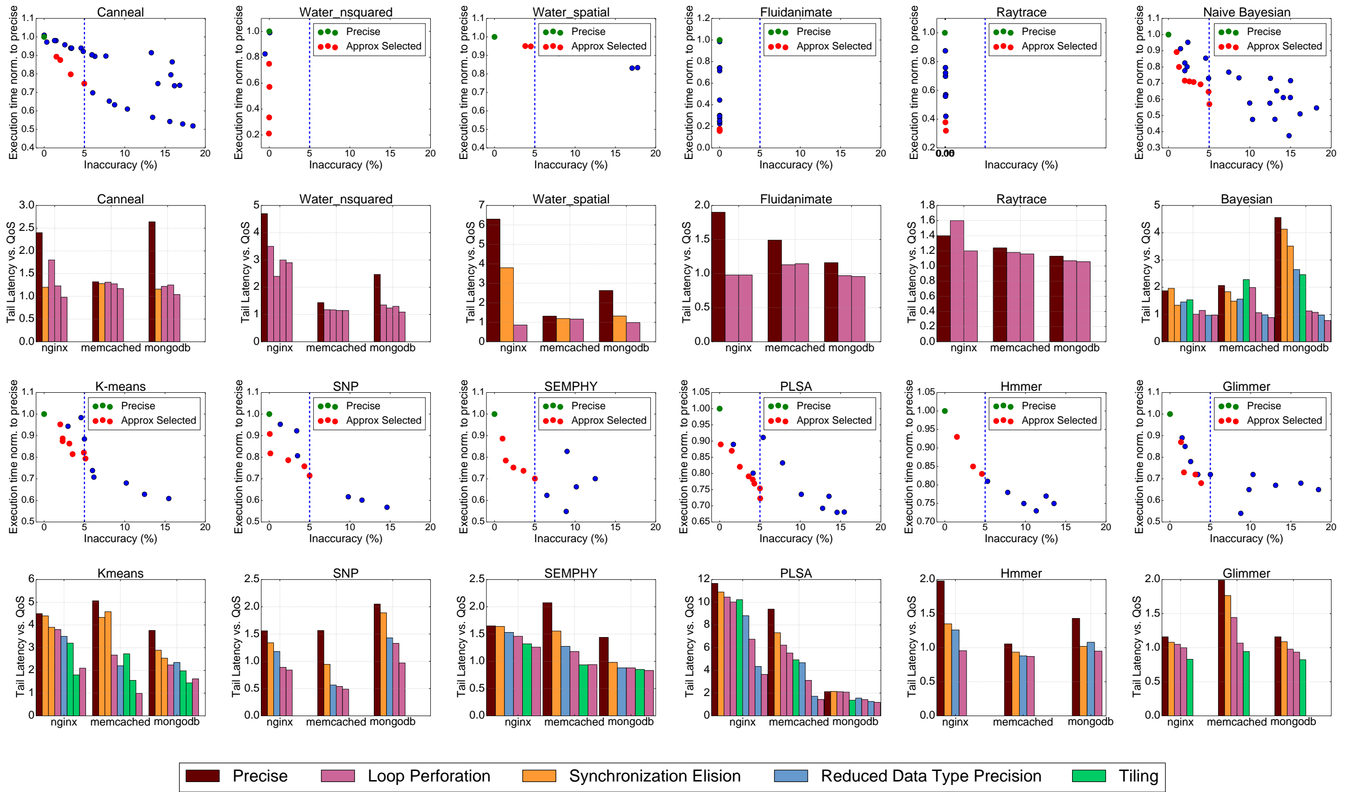

Fig. 1: Approximation design space exploration for a subset of our 24 PARSEC, SPLASH-2, MineBench, and BioPerf applications. Odd rows show the trade-off between execution time and inaccuracy for different approximate variants of each application. The green dot corresponds to precise execution, blue dots correspond to all examined approximate variants, and red dots to the selected variants close to the pareto-optimal curve. The vertical line corresponds to the max permissible loss of output quality. Even rows show the impact each of the selected approximate variants has on the tail latency of the three examined interactive applications.

quality is not always proportional to the decrease in execution time. This is because, depending on each application's logic, different loop iterations may contribute differently to output quality. For example, in the simulated annealing algorithm used in `canneal` [35], if the cost of the randomly chosen neighboring solution is not greater than the current solution, the current solution is retained and no useful work is done. Omitting such iterations decreases execution time without any observable impact on quality.

- **Synchronization elision:** Synchronization constructs, like locks and barriers, which are used to guarantee correctness can be elided at the cost of some inaccuracy in the final result [51]. Removing locks reduces the memory traffic that acquiring locks incurs, which can be significant, especially for highly contended locks. Apart from memory traffic, synchronization elision also benefits performance, as threads do not wait to synchronize, shortening execution time.
- **Lower precision data types:** This technique leverages the ability of certain applications to operate with lower precision to replace high-precision variables, such as "double", with lower precision data types, like "float" and "int". Reducing data type precision reduces both memory traffic, especially in data-intensive jobs, and execution time.
- **Tiling:** Tiling computes a single output and projects it onto the surrounding elements to create a tile, instead of computing an output for each individual element [52]. The larger the size of the tile, the more aggressive the approximation.

Pruning the design space: We now study the trade-off between accuracy and execution time for approximate computing applications, and select approximate variants close to the pareto optimal curve.

Typical applications have a large number of loops which can be perforated, or elements which can be tiled in several ways, as well as synchronization primitives that can be elided, and data types whose precision can be lowered. Considering all approximation possibilities makes the design space intractable, in the order of 1000s of approximation versions for typical applications [35]–[38]. We use two ways to prune the approximation design space. First, we employ an "almost" exhaustive exploration that leverages programmer hints from the ACCEPT framework [45]. ACCEPT lists approximately 10 loops that can be perforated for each examined application, as well as data types whose precision can be lowered and locks/barriers that can be elided. We perforate each loop by different factors, and only preserve variants with inaccuracies lower than 5%. We follow the same process for synchronization elision, and high-precision data types. Second, for applications not supported by ACCEPT we use `gprof`, an application profiling tool to determine which functions contribute the most to execution time. In all examined applications, this corresponds

to 2-4 functions, which we perforate by varying degrees. This approach also resulted in a manageable number of favorable approximate variants, consistent with those obtained using the hints from ACCEPT.

Figure 1 shows the application design space exploration for a subset of all examined applications. We use three popular cloud services, `NGINX`, `memcached`, and `MongoDB` as the latency-critical interactive applications, and 24 data mining, bioengineering, and scientific workloads as the approximate applications. We show the exploration for 12 representative approximate applications; the behavior of the remaining workloads is similar. For now we co-schedule each interactive service with one of the approximate applications at a time on a high-end two-socket server platform. More details on the applications and systems can be found in Section V.

Odd rows in the figure show the tradeoff between execution time and inaccuracy across approximate variants for each approximate application. The blue dots in the scatter plots represent all examined approximate variants, the green dot represents precise execution, and the red dots represent approximate variants close to the pareto-optimal frontier, and hereafter used by Pliant. Both the number of selected approximate versions and their relative impact on performance and inaccuracy varies across applications. For example, `canneal` has four versions close to the pareto curve, while `raytrace` only has two. Similarly, while increasing inaccuracy has an almost inversely proportional impact to execution time, all approximate versions of `water_nsquared` reside in an almost vertical line.

Even rows in the figure correspond to the impact precise execution and each of the selected approximate variants (red dots) have on the tail latency (99th percentile) of the three interactive services. Approximate variants are ordered from left to right in the way that they appear in the scatter plot. Different colors correspond to different methods of approximation. Loop perforation (at various degrees) corresponds to the majority of selected approximate variants, especially for PARSEC and SPLASH-2, while synchronization elision, tiling, and data type precision are more prevalent for Minebench and BioPerf applications, whose synchronization primitives and high-precision data types incur high levels of resource contention. First, we observe that precise execution almost always leads to considerable QoS violations across interactive and approximate applications. Second, approximation has a different impact on each of the three interactive services. While switching from precise to the least approximate variant is enough for the I/O-bound MongoDB to meet its QoS in many cases, both NGINX and memcached exhibit higher sensitivity to resource contention. This translates to requiring the approximate application to run at its most approximate variant for QoS to be met. Even so, there are approximate applications where, despite the reduction in latency from employing approximation, the latter alone is not enough to meet QoS, e.g., `kmeans-NGINX`, `PLSA-Memcached`, and `SEMPHY-NGINX`. Conversely, there are cases, e.g., `canneal-Memcached`, and `water_nsquared-Memcached` where approximation does not have a substantial impact on tail latency. These correspond to approximate versions that do not significantly decrease contention in shared resources. In these cases reclaiming resources from the approximate application is necessary to meet QoS; approximation is then used as a means to avoid prolonging the application's execution time. Finally applications like `Bayesian` and `PLSA` offer a very rich design space with 8 approximate variants on the pareto curve each; this allows the runtime to sacrifice the minimum amount of quality necessary to meet QoS at each point.

IV. PLIANT

A. Overview

Pliant consists of an *instrumentation system* that explores the approximation design space offline, and an *online runtime* that monitors performance and adjusts the degree of approximation during periods of high resource contention. Pliant's user interface involves expressing an interactive service's QoS target, and an approximate application's nominal execution time, its output quality metric, and its worst-case allowable quality loss.

The *instrumentation system* explores the various approximation techniques described in Sec. III, and obtains an ordered list of approximate variants close to the pareto frontier seen in Fig. 1. This process only needs to happen once, unless the application changes. In Sec. IV-E, we describe how Pliant handles the impact that different input datasets have on the selected approximate variants. Having the pareto frontier, the *runtime* can dynamically determine the degree of approximation needed to meet QoS at each point in time. The runtime in Pliant consists of a *performance monitor* and an *actuator* based on dynamic recompilation, as seen in Figure 2.

The *performance monitor* is a lightweight tracing runtime that instruments the interactive applications, and continuously samples their end-to-end latency (average and tail). Since QoS metrics capture the end-to-end latency, the monitor resides on the client, and is designed to not incur any measurable overhead to the interactive service, either in terms of throughput or latency. Upon detecting a QoS violation for the interactive workload, the monitor informs Pliant, which takes action via the *actuator*. The actuator is responsible for determining and enforcing the appropriate approximation variant and resource allocation at each point in time, based on the monitored tail latency. Both components are designed to incur *minimal runtime overheads* from monitoring and dynamic recompilation, to be *transparent to the user*, and to *preserve the performance of both the interactive and approximate application*, with the minimum loss in quality.

B. Dynamic Recompilation

The Pliant actuator relies on DynamoRIO [31], a dynamic recompilation tool, to adjust an application's degree of approximation at runtime. The DynamoRIO API provides the ability to control applications at the granularity of individual instructions, as well as at the coarser granularity of functions. To avoid performance overheads from instrumentation, we use DynamoRIO at coarse granularity. Specifically, we use

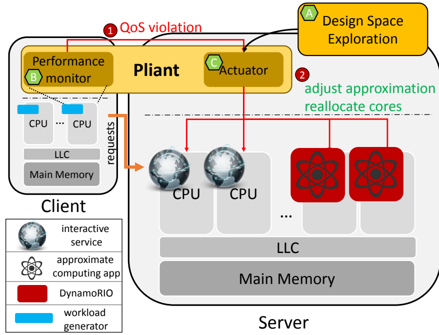

Fig. 2: Overview of Pliant's approximation space exploration, and runtime (performance monitor and actuator).

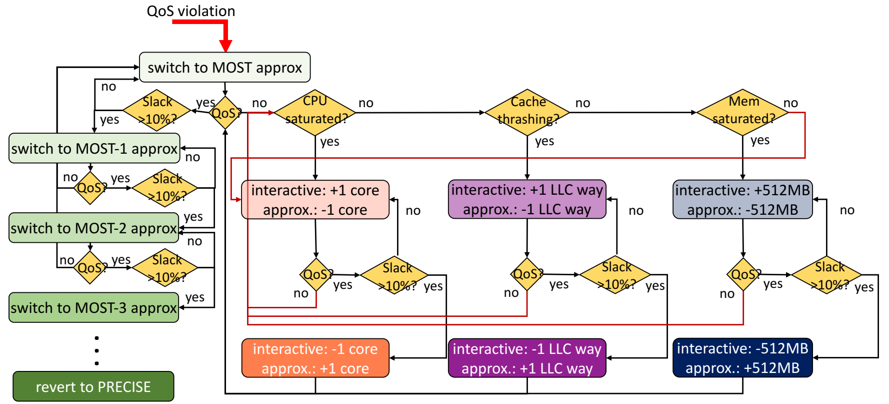

Fig. 3: Pliant's runtime when using both approximation and resource reclamation.

the drwrap_replace() interface to dynamically replace functions in the program with their approximate variants. Additionally, Pliant uses DynamoRIO's ability to trap Linux signals received by the application, to signal when a switch to/from an approximate function must occur.

Pliant first uses the approximate variants extracted from the design space exploration in Section III to construct a single application binary. This aggregates one version that corresponds to precise execution, as well as all the different versions of the functions that house the perforated loops, and versions of functions with lower precision data types, tiling, and elided synchronization primitives. Each approximate variant is then mapped to a unique Linux signal; upon receiving the specific signal, DynamoRIO switches the application to the corresponding approximate variant.

Approximate applications are executed over DynamoRIO as follows. First, DynamoRIO reads the program addresses of the precise and approximate versions for each approximated function at the start of the program. Second, during runtime, DynamoRIO traps the mapped Linux signals sent to the approximate application by the actuator. Third, when a signal is received at runtime, drwrap_replace() replaces the pointers to the original precise version of a function, to the corresponding approximate version using the program addresses read at start-up time. drwrap_replace() is also used to switch between approximate variants, or to revert back to precise execution.

Running an application over DynamoRIO can introduce overheads for the approximate applications. Across the 24 approximate applications we study, the execution time overhead is 3.8% on average, and up to 8.9% in the worst case (see Section VI for details). Prior work, such as ProteanCode [22] has shown that the overheads from tools like DynamoRIO are often prohibitively high for online code transformations, such as inserting non-temporal cache hints before the execution of certain loads to avoid cache contention. In that case, the high overhead of DynamoRIO comes from requiring code transformations to happen at the granularity of individual instructions. Because Pliant only switches between precise and approximate variants at coarse granularity, it can leverage dynamic recompilation with marginal overheads. Additionally, the overhead of dynamic recompilation in Pliant is almost always hidden by the shorter execution time of applications employing approximation to reduce resource contention.

C. Runtime Algorithm

Pliant uses the output of the performance monitor to determine the degree of approximation and resource allocation at runtime. Fig. 3 shows the control flow of Pliant's runtime algorithm. Initially execution starts at precise mode, and with a fair allocation of resources. In the event of a QoS violation, Pliant switches the co-scheduled application to its most approximate version to avoid prolonged degraded performance for the interactive service. If QoS is met in the next decision interval (1s by default), Pliant checks the latency slack of the interactive service. If the latency slack is greater than 10%, the runtime incrementally reverts back to less approximate versions - and potentially precise execution - to avoid unnecessarily penalizing the approximate application's output quality. If QoS is met, but there is not sufficient latency slack, Pliant remains in the same state (approximate or precise) for the next decision interval.

As shown in Fig. 1, there are cases where approximation alone is not enough to meet the interactive service's QoS. If the application runs in its most approximate variant and QoS is not met, Pliant additionally reclaims resources from the approximate application, incrementally until QoS is met. Pliant uses resource utilization as a trigger for the type of resource to be reclaimed. For example, if CPU utilization is saturated, Pliant reclaims cores, one per interval, until QoS is met, or until another resource is saturated. Similarly, if the memory utilization of the interactive service is saturated, Pliant resizes memory allocation of the approximate application's container, and yields the reclaimed memory to the interactive service. Finally, if the LLC experiences high miss rates, Pliant leverages cache partitioning via Intel's Cache Allocation Technology (CAT) to resize the LLC partition of the interactive and approximate applications. Once QoS is met, Pliant checks again for latency slack. If slack is greater than 10%, the runtime reverts to the previous state by returning the reclaimed resource to the approximate application. If slack remains high, the runtime additionally decreases approximation to the minimum

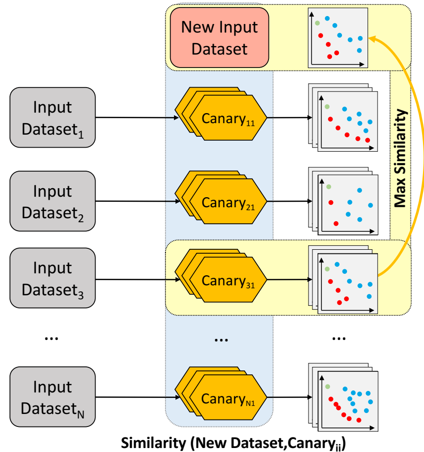

Fig. 4: Approximation design space exploration using canary inputs and input similarity.

TABLE I: Platform Specification

Model	Intel Xeon E5-2699 v4
OS	Ubuntu 16.04 (kernel 4.14)
Sockets	2
Cores/Socket	22
Threads/Core	2
Base/Max Turbo Frequency	2.2GHz / 3.6GHz
L1 Inst/Data Cache	32 / 32 KB
L2 Cache	256KB
L3 (Last-Level) Cache	55 MB, 20 ways
Memory	16GBx8, 2400MHz DDR4
Disk	1TB, 7200RPM HDD
Network Bandwidth	10Gbps

needed to meet QoS. Finally, if during runtime, the application is operating at an approximation degree other than the highest and a QoS violation occurs, it immediately reverts to its most approximate variant.

Varying the slack threshold affects Pliant's agility in adjusting resource allocations and approximation degrees. Lowering the threshold further results in frequent ping-ponging between states, and higher overheads from DynamoRIO. Relaxing the threshold does not have an impact on the interactive service, but can degrade the performance and/or quality of the approximate application, when running with a higher approximation degree, or fewer resources than necessary. Unless otherwise specified, we use a 10% latency slack threshold.

D. Multi-Application Colocations

So far we have assumed that an interactive service is colocated with a single approximate application. To increase utilization, servers often co-schedule multiple jobs per physical host, especially when each task is short [53]. We now extend Pliant to handle more than one approximate application per machine. The system starts again from a fair resource allocation, and all approximate applications operating in precise mode. When a QoS violation is detected, Pliant manages the approximate applications in a round-robin fashion, to avoid penalizing any of the applications in a disproportionate way. It first switches one workload (selected randomly) to its most approximate variant, and if QoS is not restored it moves to the next. If all applications operate in their most approximate variants and QoS is still not met, Pliant reclaims resources from the approximate applications, one application at a time until QoS is met, following the runtime algorithm discussed in Fig. 3. The round-robin arbiter is simple, scalable, and preserves all co-scheduled applications' performance in practice (see Sec. VI); in Section VI-E we discuss alternative policies to manage multiple approximate jobs.

E. Robustness to Input Datasets

Changing the characteristics of an application's input dataset can impact the shape of its performance-accuracy pareto frontier, and the selected approximate variants. In such cases, the exploration would need to happen for each different type of input dataset, which can incur higher management overheads.

Recent work has shown that the approximation trade-offs of an application's input dataset can be accurately captured by smaller, canary inputs, which accelerate the exploration process [54]. Canaries are created using regular, strided subsets of the full input of size N/16, N/32, N/64, N/128 and N/256 where N is the size of the full input. For example, for inputs in the form of 1D lists of numbers or structs, we pick every 16th element for an input of size N/16. For 2D inputs, we pick every $(1/\sqrt{t}, 1/\sqrt{t})$ element for size N/t.

We implement a similar approach to accelerate the design space exploration in Pliant. Fig. 4 shows an overview of this process. We initially create a set of diverse input datasets for each approximate application. Datasets differ with respect to their size, locality, and other application-specific parameters. We then create canary inputs for each original inputs, and perform the design space exploration using the canaries.

At runtime, when a previously-seen application runs with a new, unknown input dataset, Pliant computes the similarity of the new dataset with all existing canaries using the methodology proposed in [54], and selects the approximate variants of the canary with the highest similarity to the new input. We use the F-test for equality of variances to find the similarity between the full input and the canaries. We select random samples from the canaries and the new input and compute the test statistic (t-value) for each canary, i.e., the ratio of variance of the data from the canary to the full input. Finally, we calculate p-values using the F-distribution over the test statistic and the Holm-Bonferroni [55] method to find the canaries that are most similar to the input. As in [54], we use canary error bounds $\alpha = \beta = 0.05$. This eliminates the need for design space exploration when at least one canary captures the features of the input dataset. If no existing canary accurately reflects the new dataset, Pliant creates canaries for this input, and conducts the design space exploration on the most similar canary. To avoid repeating this process in the future, the new input is added to the repository of previously-seen datasets for the given application, as is the set of its canaries, and the output of their characterization. This allows Pliant to learn a wider spectrum of input behaviors over time, and progressively decreases the need for online design space exploration.

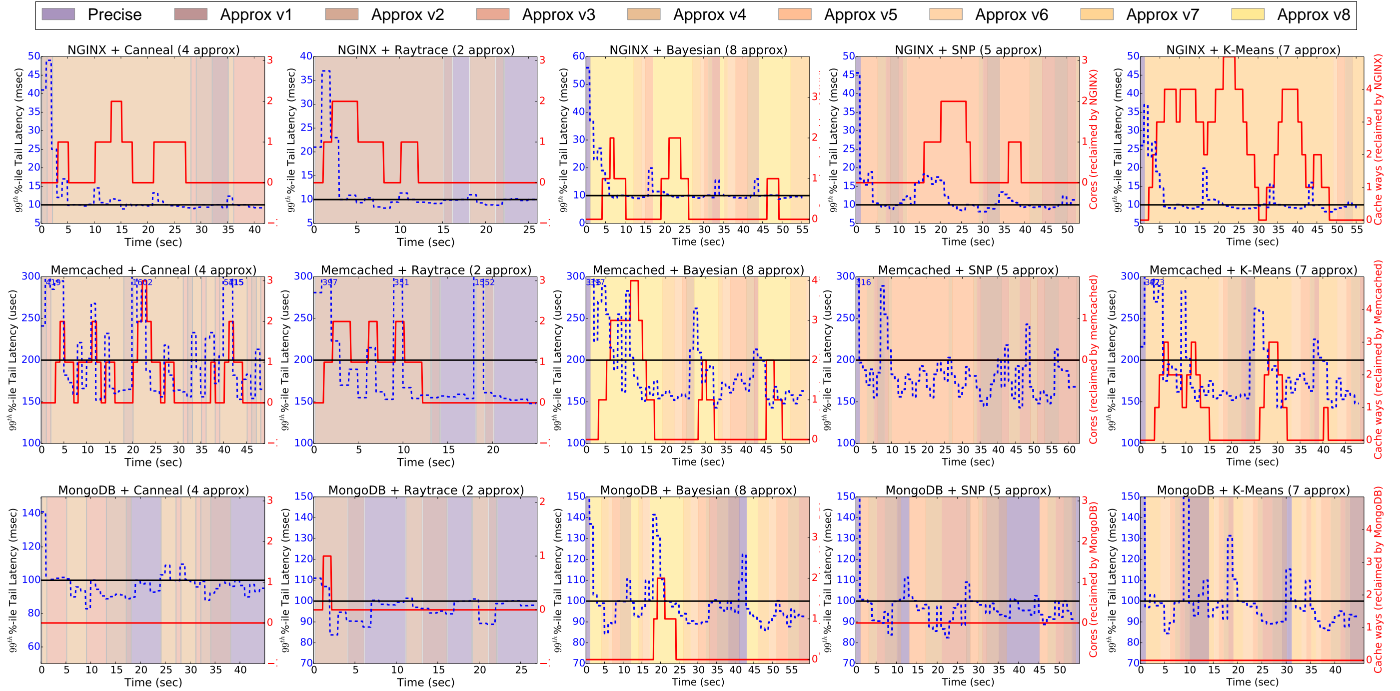

Fig. 5: Pliant's dynamic behavior when each of the three interactive services (one per row) are colocated with selected approximate computing workloads. The left y-axes show the tail latency of the interactive service over time (dashed blue line), and the right y-axes the cores Pliant reclaims from the colocated approximate application for the 4 left-most workloads (red line). For the last memory-intensive approximate workload, we show LLC ways reclaimed and given to the interactive service using CAT. The black horizontal line corresponds to the QoS constraint of each interactive service. The colored background reflects the approximate version Pliant employs at each point in time. Darker colors correspond to versions closer to precise execution, and lighter colors implement more aggressive approximation, within the 5% permitted inaccuracy threshold.

V. Experimental Methodology

Interactive services: We use three latency-sensitive applications, `NGINX`, `memcached`, and `MongoDB`.

- `NGINX` [33] is a high-performance HTTP webserver, and is currently responsible for 38% of all live websites as of April 2018 [56]. We use `NGINX` as a front-end webserver to display static HTML files. The input dataset consists of one million unique HTML files of 1KB each. The QoS target for `NGINX` is determined as the 99th percentile latency before the knee of the latency-throughput curve when the application runs in isolation, and is set at 10ms, consistent with related work [8], [57]–[60].
- `Memcached` [32] is an in-memory key-value store, often used as an object caching tier in cloud services [59], [61], [62]. We configure its dataset to hold 5 million items, each with 30B key, and 200B value. The QoS target for memcached is defined using the same process as above, and set to 200us, consistent with prior work [8], [9].
- `MongoDB` [34] is one of the most popular NoSQL databases, and is widely used in industry for back-end data storage [60], [63]. We use MongoDB 3.2.16 compiled from source, and compose a dataset with 160 million records, each with 10 fields and 100B per field. The dataset is 178GB, including indices and metadata, and the QoS is 100ms.

All interactive services are driven by open-loop load generators. We instantiate enough clients to avoid client-side saturation, and ensure that most latency is due to server-side delay. Unless otherwise specified, we run the interactive services at high load, approximately 75-80% of saturation.

Approximate computing applications: We use 24 data mining, bioengineering, and high performance computing applications from four suites as the approximate computing applications. Specifically, we use three workloads (`fluidanimate`, `canneal`, `streamcluster`) from PARSEC [35], three workloads (`water_spatial`, `water_nsquared`, `raytrace`) from SPLASH-2 [36], ten applications (`Naive Bayesian`, `K-means`, `SEMPHY`, `Fuzzy-K-means`, `BIRCH`, `SNP`, `GeneNet`, `SVM-RFE` and `PLSA`, `ScalParC`) from the Minebench benchmark suite [38] and 8 applications (`Hmmer`, `Blast`, `Fasta`, `GRAPPA`, `ClustaLW`, `T-Coffee`, `Glimmer`, `CE`) from the Bioperf benchmark suite [64]. All selected applications have metric(s) to quantify the quality of their output, and have been previously shown to tolerate some loss in their quality for improved performance and/or efficiency [26], [28], [45]. We select workloads from several fields to ensure good coverage of the approximate computing space, and to show how Pliant behaves under different application characteristics.

Systems: We use a dual-socket, 44-physical core (88 logical core) platform, with 128GB of RAM as the server. Table I summarizes the specification of our experimental platform. To avoid NUMA effects, we only use one of the sockets

for the interactive service, and the approximate applications. The interactive service, and the approximate workloads are instantiated in separate Docker containers, and pinned to different physical cores of the same socket. The containers share the 56MB last level cache (LLC), the main memory, and the NIC. Additional six physical cores are dedicated to network interrupts (soft_irq) to avoid interference with application threads. The remaining physical cores are fairly shared across the two Docker containers. For now we assume a single approximate workload co-scheduled with an interactive service. In Section IV-D we also discuss how Pliant treats multiple approximate applications colocated with a latency-critical service. In that case the total available resources are again fairly shared among all applications at start up time.

VI. Evaluation

We first evaluate Pliant's behavior for a few representative approximate applications, then show its performance and efficiency across all studied applications, and finally show the runtime's sensitivity to configuration parameters.

A. Dynamic Behavior

Figure 5 shows Pliant's dynamic behavior when each of the three interactive services is colocated with one of four selected approximate workloads. We select workloads that exhibit diverse characteristics with respect to their resource requirements, performance sensitivity, and number and effectiveness of approximate variants. By default Pliant uses a one second decision interval at the end of which, it makes a decision on the degree of approximation and resource allocation needed for the two applications. In Section VI-D we study the impact of varying the decision granularity. When a QoS violation is detected, the runtime switches the colocated application to its most approximate variant, and if that is not sufficient, it additionally reclaims resources and yields them to the interactive service, incrementally once per decision interval. To avoid penalizing the approximate application when the interactive service has a lot of latency slack, Pliant also returns resources to the approximate workload when slack exceeds 10%. The four left-most approximate applications in Fig. 5 only experience core reclamation, while k-means also has its LLC partition resized.

We first examine applications in the same column. When canneal is co-scheduled with NGINX it almost immediately has to switch to the most approximate of its four versions, due to high compute and cache contention, and additionally relinquish 1-2 cores to the interactive service. As the tail latency of NGINX drops, canneal reclaims these cores, and additionally switches to an implementation variant closer to precise towards the end of its execution. memcached experiences even higher sensitivity to resource contention, forcing canneal to operate in its most approximate variant for the majority of execution, and yield up to 3 cores to address short bursts of high tail latency. In contrast, the I/O-bound MongoDB needs no additional cores to meet its QoS target, and even enables canneal to run at precise mode for significant periods of its execution. We observe similar trends for the three interactive services, across the other approximate applications shown in Fig. 5. For example, only raytrace and Bayesian have to yield cores for brief periods of time when co-scheduled with MongoDB.

We now examine applications in the same row of the figure. Compared to canneal, raytrace only has two possible approximate variants. Because raytrace only introduces high compute and LLC interference in certain execution phases, it is able to leverage both its approximate variants, as well as precise execution across all three interactive services. bayesian offers a much richer design space with eight approximate variants close to the performance-quality pareto curve (Fig. 1). This allows Pliant to make frequent, fine-grained decisions that only sacrifice the minimum amount of output quality needed at each point in time. The figure shows that when bayesian is running with NGINX and it has not yielded any cores to the interactive service, tail latency is closely correlated with the approximate variant bayesian uses, e.g., as bayesian switches to decreasingly approximate versions in $t \in [26, 34]$, tail latency increases until it exceeds QoS, at which point bayesian returns to its more approximate version (lightest background color in the graph). SNP is the only of the four approximate applications pictured that enables both memcached and MongoDB to meet their QoS throughout the duration of the experiment using approximation alone. This happens because SNP's approximate variants employ synchronization elision and perforation, and are particularly effective at reducing the amount of contention in the shared LLC. SNP only has to relinquish up to two cores when co-scheduled with NGINX.

Finally, k-means initially only employs approximation to meet the interactive service's QoS. As the scenario progresses and cache contention increases, it additionally yields several of its last-level cache ways to the interactive workload. NGINX and memcached experience the highest sensitivity to cache resources, with MongoDB only requiring k-means to run in approximate mode to meet its QoS requirements, without changing the initial fair cache allocation.

Constraining the allocated resources does not translate to a performance penalty for the approximate applications, with all five of them achieving equal or better performance compared to precise execution, and a 2.7% average loss in output accuracy.

B. Pliant Generality

We now evaluate Pliant across all 24 examined approximate applications and 3 interactive services. The decision interval is again one second. Figure 6 shows the $99^{th}-ile$ tail latency, execution time, and inaccuracy for the baseline system (Precise), a runtime that enforces precise execution but uses resource reclamation, and Pliant. The bars show tail latency for both runtimes, and the markers execution time for the approximate applications. The black bars show the tail latency of each interactive service, when running in isolation. The marker labels show the loss of output quality as a result of employing approximation with Pliant. The baseline precise system always

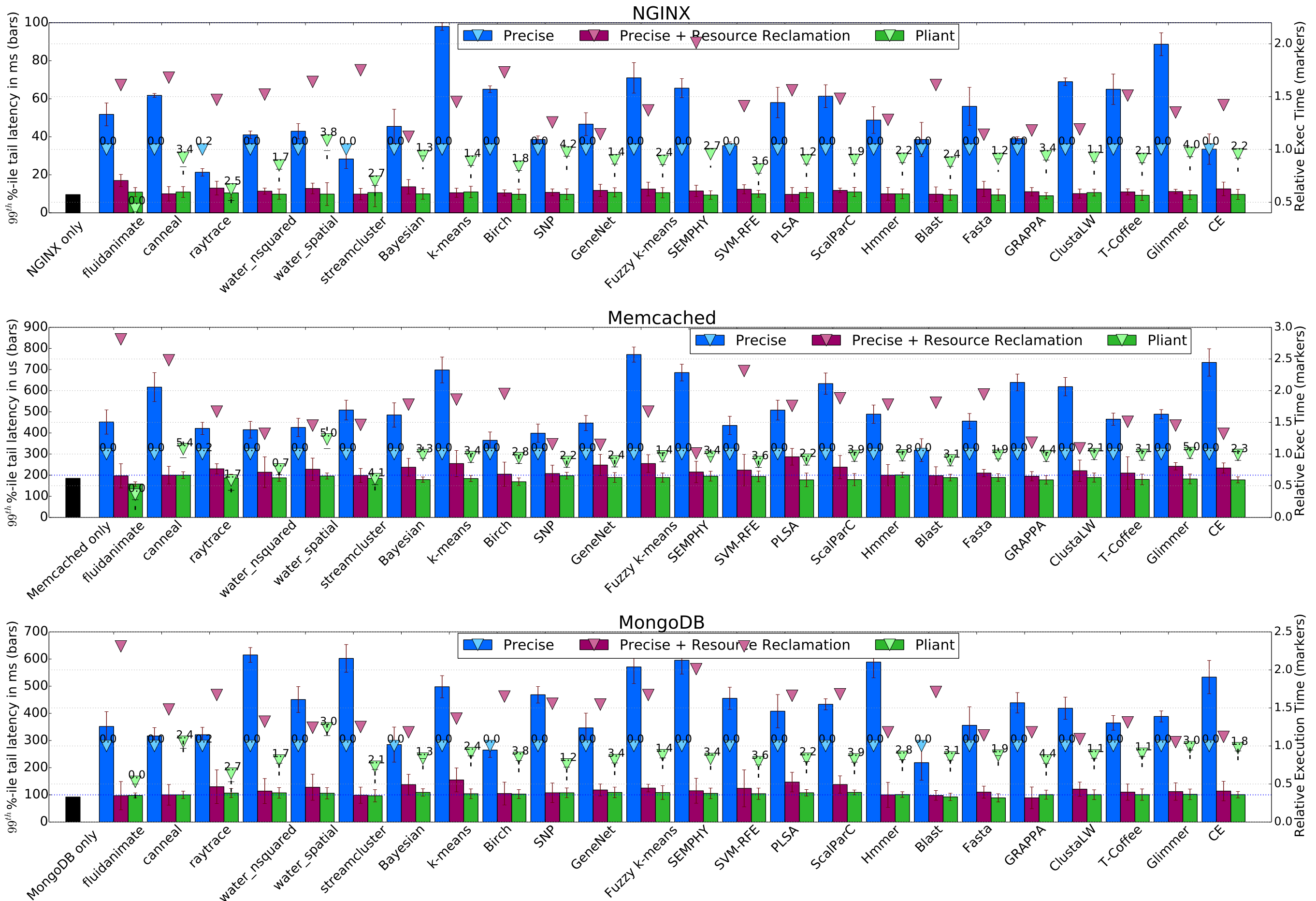

Fig. 6: Comparison of Pliant against the Precise runtime, and the Precise runtime with resource reclamation for all interactive and approximate applications. The tail latency of the interactive services is shown in bars, while markers represent the execution time of approximate applications. The marker labels denote the % inaccuracy. The error bars denote tha variance across 10 runs of each experiment. The QoS targets for `NGINX`, `memcached`, and `MongoDB` are 10ms, 200us, and 100ms respectively. The single black bars in each plot show the tail latency of the interactive application when running in isolation.

operates with nominal accuracy. Additionally, in the baseline system, both the interactive service and the approximate application receive a fair resource allocation. Running in precise mode always results in severe QoS violations for the interactive service, $2.1-9.8x$ for `NGINX`, $1.46-3.8x$ for `memcached`, and $2.08-5.91x$ for `MongoDB`. In comparison, Pliant meets QoS for each of the interactive services across the 24 colocated approximate applications. Additionally, all approximate workloads, except for `water_spatial`, maintain their nominal performance (precise execution), and in several cases improve it. In the case of `water_spatial`, the selected approximate variants do not significantly reduce its execution time, and its decreased resource allocation results in higher execution time than in precise mode. `water_spatial` also experiences an unusually high instrumentation overhead from DynamoRIO (seen from the whisker in Fig 6), which also contributes to its execution time. The precise execution with resource reclamation meets the interactive service's QoS for the most part, but heavily penalizes the performance of the approximate application. Furthermore, in some cases QoS is not met, since precise execution can result in higher contention in resources like memory bandwidth and cache capacity, even with a reduced resource allocation. Such applications typically employ synchronization elision and reduced data type precision in Pliant to reduce the incurred interference.

The overhead from DynamoRIO is 3.8% in execution time on average, and up to 8.9%. The reason behind the low overhead is that dynamic instrumentation is invoked at coarse granularity, as opposed to instruction-level transformations [22]. Finally, the loss in output quality is 2.1% on average, and within the 5% tolerable limit for all applications, except for `canneal` when colocated with `memcached`. In that case inaccuracy is 5.4%, due to some non-determinism caused by synchronization elision.

C. Multi-Application Colocations

We now evaluate the case where Pliant handles more than one approximate applications sharing a physical host with an interactive service. In that case Pliant examines approximate applications in a round-robin fashion to ensure that no approximate application is penalized disproportionately.

Selected colocations: Figure 7 shows examples of two approximate applications at a time sharing a server with each

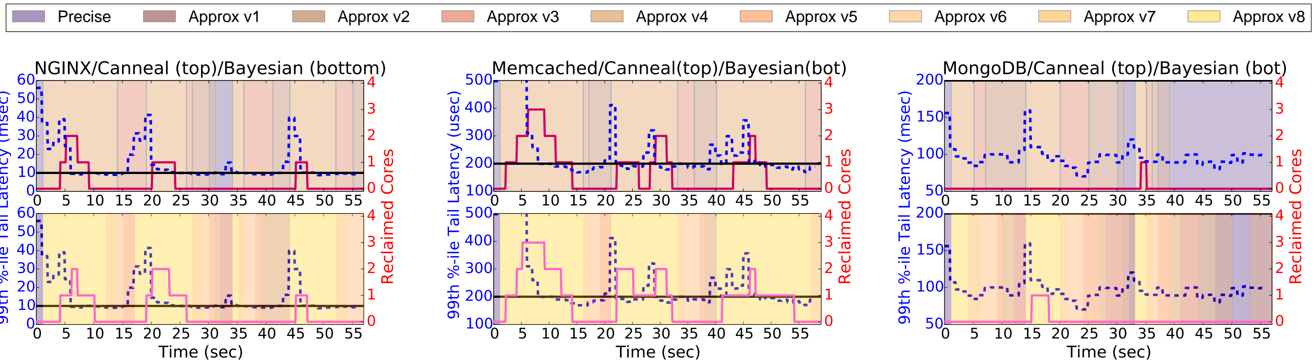

Fig. 7: Pliant managing colocations with multiple approximate applications at a time (*canneal*, and *bayesian* in this case). The top graph shows the approximate variants and resources reclaimed from `canneal`, while the bottom graph shows the same for `bayesian`.

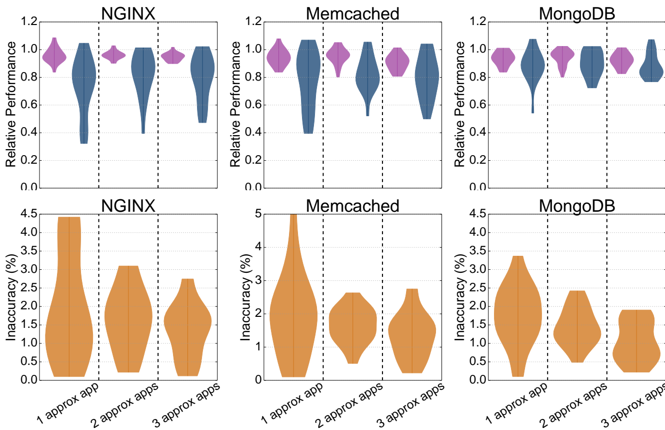

Fig. 8: Violin plots of tail latency for the interactive service (purple), execution time (blue) and inaccuracy (orange) for the approximate workload across colocations with 1, 2, and 3 approximate applications for the three interactive services. Tail latency is normalized to QoS, and execution time normalized to precise execution. The limits of the violins show the min and max value of each metric.

interactive service. For simplicity, we focus on cases where cores are reclaimed from the approximate application; the process is the same for other reclaimed resources. The top figure shows the approximate variants of `canneal` over time, and the cores Pliant reclaims and yields to the interactive service. The bottom figure shows the same metrics for `bayesian`. The interactive service's tail latency is shown in both figures. Unlike when `canneal` or `bayesian` alone were colocated with `NGINX`, in which case multiple cores had to be reclaimed for `NGINX` to meet its QoS, now each application at most yields one core. Therefore for a large fraction of execution, approximation alone is sufficient to meet QoS. This enables both applications to keep their quality loss low, and preserve their nominal performance.

As before, `memcached` is more sensitive to interference than the other interactive services. This results in employing more aggressive approximate variants, and reclaiming more cores from both approximate workloads. In contrast, `MongoDB` rarely needs additional cores, while towards the end of the scenario it can meet its QoS while `canneal` operates in precise, and `bayesian` in near-precise mode. Note that there is no case where a single application sacrifices a disproportionate amount of its accuracy or resources.

Aggregate results: We now generalize the previous experiment across all studied interactive and approximate applications. Figure 8 shows violin plots of tail latency for the interactive service (purple), and execution time (blue) and loss of output quality (orange) for the approximate workloads, when we colocate each interactive service with one, two, and three approximate applications at a time. The limits of the violins capture the min and max value of each metric. We examine all 2- and 3-way application combinations of the 24 approximate workloads. Across all three interactive services, as we increase the number of colocated applications the violins of inaccuracy become more centralized. This is consistent with Fig. 7 which shows that all colocated approximate applications sacrifice comparable amounts of their output quality. In comparison, when a single approximate application shares a node with an interactive workload it may have to sacrifice considerable quality to meet the interactive service's QoS, although without exceeding its 5% allowed threshold. The execution time violin plots for the approximate workloads reveal a similar trend of less diverse performance as consolidation increases.

Across interactive services, `MongoDB` incurs the lowest impact on the approximate workloads, both in execution time and inaccuracy, since in many cases applications operate in precise mode, without impacting `MongoDB's` tail latency.

D. Pliant Sensitivity

Input load: Fig. 9 shows tail latency for each interactive service, and execution time and inaccuracy for each approximate workload, as we vary the input load (QPS) of the interactive service. We focus on colocations with a single approximate workload for clarity, and examine loads between 40% to 100% of saturation in increments of 10%. Lowering the load further has no impact on either tail latency or execution time. When load is below 60% each of the interactive services can satisfy its QoS, while the approximate workload operates mostly in precise mode. `MongoDB` is an especially amenable co-runner, allowing colocated applications to operate in precise mode until it reaches 80-85% load.

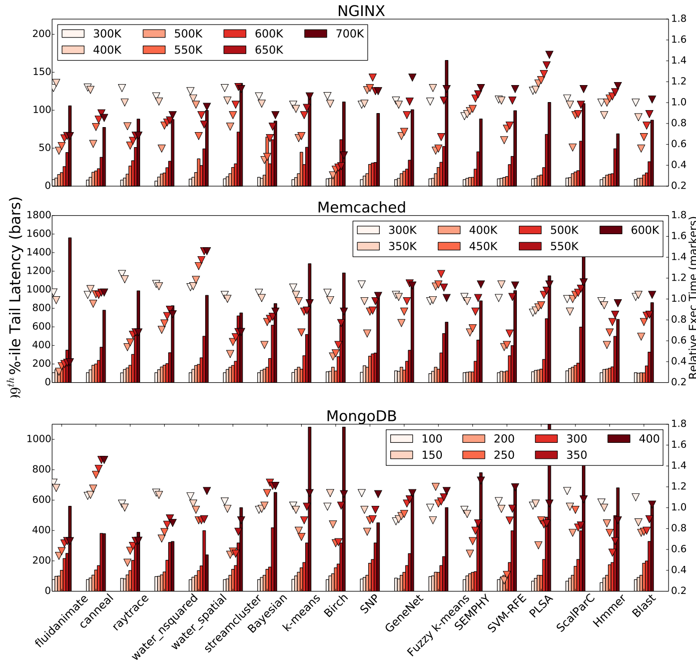

Fig. 9: Performance of Pliant across load levels (QPS) for each interactive service. The tail latency of each interactive service is shown in bars (in ms for `NGINX` and `MongoDB`, and us for `memcached`), while markers represent the execution time of approximate applications.

For `NGINX` and `memcached`, when load is between 60-70% approximation alone is often enough to meet QoS, with `memcached` requiring some applications to additionally yield 1-2 cores, or 1-3 cache ways. When load is 70-80%, approximation together with reclamation is needed to meet QoS, while increasing the load beyond 90% results in significant QoS violations regardless of the use of approximation. When the same applications operate in precise-only mode, QoS can only be met until 340K QPS for `NGINX` (48% load), 280K QPS for `memcached` (46% load), and 310 QPS for `MongoDB` (77% load). The execution time of the approximate workloads exhibits two trends. First, there are applications like `water_spatial` for which increasing the load of `NGINX` results in progressively shorter execution time because the degree of approximation increases, without the need for resource reclamation. On the other hand, there are applications like `PLSA`, where increasing the load results in a slight increase in execution time because in addition to approximation, resources must be reclaimed to meet QoS. Most applications experience both trends with execution time first decreasing, while approximation alone is used, and then increasing when resources are reclaimed.

Decision interval: Figure 10 shows the tail latency and relative execution time when we vary Pliant's decision interval. The marker labels show the loss of output quality for the approximate workload. For brevity, we show a few representative approximate applications co-scheduled with `memcached`; the trend is the same for the remaining workloads. When the decision interval is too coarse (above 1s), the interactive service experiences prolonged QoS violations until Pliant takes action. Decision intervals of 1s or less allow Pliant to satisfy QoS without penalizing the colocated application's execution time or accuracy beyond its allowed threshold. In theory, very short decision intervals can result in higher execution times for the approximate applications, due to frequent switching between precise and approximate versions. In practice, because Pliant is lightweight, no such degradation is observed. In fact in the case of `raytrace` there is the reverse trend; the application has higher execution time with longer decision intervals, because the runtime forces it to run with a smaller amount of resources than needed to satisfy QoS.

Inaccuracy threshold: For simplicity, we have so far assumed that the inaccuracy threshold is 5% for all applications. In practice, this threshold is specified by the user, and can differ per application. Fig. 11a shows Pliant's sensitivity to the inaccuracy threshold. Throughput is in kQPS for `NGINX` and `memcached`, and in QPS for `MongoDB`. Changing the threshold affects the selected approximate variants used at runtime. The x-axis shows the inaccuracy threshold, while the box plots show the distribution of maximum sustained load for each interactive service without QoS violations. The lower the threshold, the less flexibility Pliant has when employing approximation. This results in lower sustained QPS for the interactive service, with `NGINX` and `memcached` experiencing the highest throughput degradation, while `MongoDB` remains mostly unchanged for thresholds above 2%. As the inaccuracy threshold increases beyond 5%, Pliant has more leeway when trading off accuracy for performance. In this case, all three interactive applications reclaim more resources from the approximate workloads, which translates to higher QPS.

Breakdown of effectiveness: Finally, Figure 11b shows the fraction of colocations for which approximation alone was sufficient to meet the QoS of each interactive service, versus cases where different resources have to be reclaimed, across the application's runtime. Cases where multiple types of resources have to be reclaimed at a time, are shown in the stacked bar chart. This includes 1-, 2-, and 3-approximate application mixes co-scheduled with an interactive service. When approximation alone is not sufficient, resources (cores, cache, or memory capacity) are reclaimed from the approximate workloads. The "0 cores" bars correspond to cases where only memory resources (cache and main memory) are reclaimed. The "only" bars correspond to cases where only N cores are reclaimed, where N is shown in the x-labels. In the case of `NGINX`, for 33% of experiments, approximation was sufficient to resolve any QoS violations, without constraining the resource allocation of the approximate applications. The majority of these experiments correspond to single approximate application scenarios, since in that case the interactive service starts with a larger fraction of system resources.

A smaller fraction of experiments results in 1-2 cores being reclaimed from the approximate applications, especially for applications where approximation itself does not reduce resource contention. Cache and memory reclamation is rare for `NGINX`; similarly reclaiming 3 or more cores is unlikely in practice. The results differ for `memcached` and `MongoDB`.

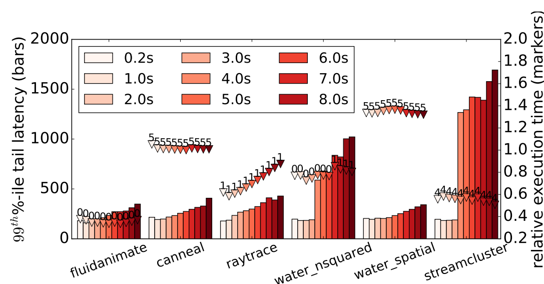
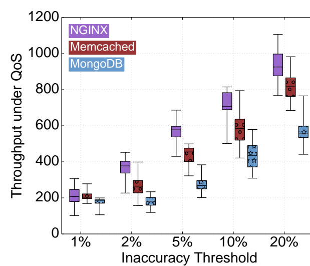
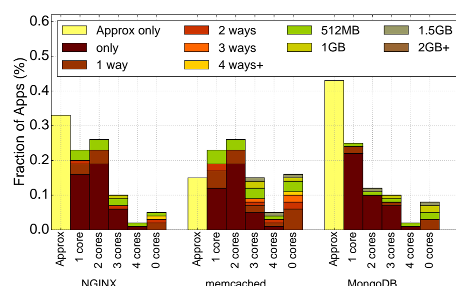

Fig. 10: Tail latency of memcached, and execution time for selected approximate workloads when varying the decision interval.

Fig. 11: (a) Pliant's sensitivity to the max inaccuracy threshold. (b) Breakdown of cases where approximation alone is enough to meet QoS, versus cases where resources have to be reclaimed from the approximate workload(s).

Unlike NGINX, memcached almost always requires at least one core to be reclaimed from the approximate applications, and often cache and memory capacity as well, primarily due to its strict QoS which makes it sensitive to resource interference. MongoDB, on the other hand, is the most amenable of the three interactive services, and can meet its QoS leveraging approximation alone, or one reclaimed core or cache way in the majority of cases. This information can be incorporated in the cluster scheduler when deciding which applications to place on the same physical node.

E. Limitations

Pliant can be extended in several ways. First, even though considering multiple approximate applications in a round-robin fashion provides a simple form of arbitration and is effective in practice, more sophisticated policies can offer better performance and/or resource efficiency. For example, considering the relative performance impact of approximation across co-scheduled applications can prioritize adjusting the quality and/or resources of the application that will be hurt the least. Second, even though Pliant uses canary inputs to reduce the overhead of the design space exploration, there are still cases where this process needs to occur for a new application. This may not always be possible, especially in the context of public clouds, where the cloud provider does not have source code access to the end user's applications. In that case, the user can provide the approximate variants, or hints on primitives that can be approximated using a framework like ACCEPT, and the relative impact of approximate versions can be learned at runtime [29], [45]. In contrast, in Software-as-a-Service (SaaS) and serverless cloud settings where the user leverages the cloud's applications via fine-grained functions, the provider has source code access to perform the exploration Pliant needs.

VII. Conclusions

We presented Pliant, a practical and lightweight cloud runtime that leverages the ability of approximate computing applications to tolerate some loss of output quality, to preserve the QoS of co-scheduled interactive services. Pliant relies on a lightweight performance monitor to track QoS violations, and a dynamic recompilation system to adjust the degree of approximation online. We showed that approximation exposes a wide spectrum of operating points in terms of execution time and inaccuracy, and demonstrated that Pliant can navigate this space effectively, and preserve QoS, while using the lowest degree of approximation needed across a diverse set of interactive and approximate applications.

References

[1] L. Barroso and U. Hoelzle, *The Datacenter as a Computer*. 2009.
[2] "Amazon ec2." http://aws.amazon.com/ec2/.
[3] "Google container engine." https://cloud.google.com/container-engine.
[4] L. Barroso, "Warehouse-scale computing: Entering the teenage decade," *ISCA Keynote, SJ, June 2011*.
[5] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
[6] J. Mars and L. Tang, "Whare-map: heterogeneity in "homogeneous" warehouse-scale computers," in *Proc. of ISCA*, 2013.
[7] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for qos-aware clouds," in *EuroSys*, 2010.
[8] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-Efficient and QoS-Aware Cluster Management," in *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
[9] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42st Annual International Symposium on Computer Architecuture (ISCA)*, 2015.
[10] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing (SOCC)*, August 2015.
[11] C. Delimitrou and C. Kozyrakis, "Bolt: I Know What You Did Last Summer... In The Cloud," in *Proc. of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
[12] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in *Proceedings of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, Minneapolis, MN, 2014.
[13] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini, "Deepdive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. of ATC*, 2013.
[14] 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 *Proc. of the IEEE 3rd International Conference on Cloud Computing (CLOUD)*, Miami, FL, 2010.
[15] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *Proc. of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Pittsburgh, PA, 2010.
[16] L. Tang, J. Mars, and M.-L. Soffa, "Compiling for niceness: Mitigating contention for qos in warehouse scale computers," in *Proceedings of CGO*, San Jose, CA, 2012.

[17] S. M. Zahed and B. C. Lee, "Ref: Resource elasticity fairness with sharing incentives for multiprocessors," in *Proceedings of ASPLOS*, Salt Lake City, UT, 2014.

[18] C. Delimitrou and C. Kozyrakis, "HCloud: Resource-Efficient Provisioning in Shared Cloud Systems," in *Proceedings of the Twenty First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2016.

[19] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, vol. 30, pp. 8–19, July 2010.

[20] C. Reiss, A. Tumanov, G. Ganger, R. Katz, and M. Kozych, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of SOCC*, 2012.

[21] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *Proceedings of ASPLOS*, 2013.

[22] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *Proc. of MICRO*, 2014.

[23] R. Nathuji, C. Isci, and E. Gorbatov, "Exploiting platform heterogeneity for power efficient data centers," in *Proceedings of ICAC*, Jacksonville, FL, 2007.

[24] H. Kasture and D. Sanchez, "Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads," in *Proc. of ASPLOS*, 2014.

[25] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *Proc. of MICRO*, 2015.

[26] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. of MICRO*, 2013.

[27] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[28] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," in *Proc. of OOPSLA*, 2014.

[29] S. Misailovic, "Accuracy-aware optimization of approximate programs," in *Proc. of CASES*, 2015.

[30] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard, "Proving acceptability properties of relaxed nondeterministic approximate programs," in *Proc. of PLDI*, 2012.

[31] "DynamoRIO: Dynamic Instrumentation Tool Platform." http://www.dynamorio.org.

[32] "Distributed caching with memcached," in *Linux Journal, 2004*.

[33] "Nginx." https://nginx.org/en.

[34] "mongodb." https://www.mongodb.com.

[35] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[36] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, (New York, NY, USA), pp. 24–36, ACM, 1995.

[37] A. Jaleel, M. Mattina, and B. L. Jacob, "Last level cache (llc) performance of data mining workloads on a cmp - a case study of parallel bioinformatics workloads," in *Proceedings of HPCA*, Austin, Texas, 2006.

[38] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary, "Minebench: A benchmark suite for data mining workloads," in *Proceedings of the 9th IEEE International Symposium on Workload Characterization (IISWC)*, San Jose, California, 2006.

[39] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, 2009.

[40] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *Proc. of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2011.

[41] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, 2012.

[42] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, 2014.

[43] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "homogeneous"; warehouse-scale computers: A performance opportunity," *IEEE Comput. Archit. Lett.*, vol. 10, July 2011.

[44] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Proceedings of PLDI*, (New York, NY, USA), pp. 164–174, ACM, 2011.

[45] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," in *UW-CSE-15-01-01*, 2015.

[46] W.-K. Chen, S. Lerner, R. Chaiken, and D. M. Gillies, "Mojo: A dynamic optimization system," 04 2018.

[47] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with adapt," in *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, (New York, NY, USA), pp. 93–102, ACM, 2001.

[48] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The performance of runtime data cache prefetching in a dynamic optimization system," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 180–, IEEE Computer Society, 2003.

[49] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu, "Design and implementation of a lightweight dynamic optimization system," *J. Instruction-Level Parallelism*, vol. 6, 2004.

[50] W. Zhang, B. Calder, and D. M. Tullsen, "An event-driven multithreaded dynamic optimization framework," in *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pp. 87–98, Sept 2005.

[51] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, (New York, NY, USA), pp. 41–50, ACM, 2012.

[52] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of ASPLOS*, (New York, NY, USA), pp. 35–50, ACM, 2014.

[53] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi2: Cpu performance isolation for shared compute clusters," in *Proceedings of EuroSys*, Prague, Czech Republic, 2013.

[54] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: Using canary inputs to dynamically steer approximation," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, (New York, NY, USA), pp. 161–176, ACM, 2016.

[55] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[56] "Usage statistics and market share of NGINX for websites." https://w3techs.com/technologies/details/ws-nginx/all/all.

[57] S. Chen, S. Galon, C. Delimitrou, S. Manne, and J. F. Martnez, "Workload Characterization of Interactive Cloud Services on Big and Small Server Platforms," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, October 2017.

[58] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of SIGMETRICS*, London, UK, 2012.

[59] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proc. of EuroSys*, 2014.

[60] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, England, UK, 2012.

[61] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[62] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, OS, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[63] "MongoDB official website." http://www.mongodb.com.

[64] D. A. Bader, Y. Li, T. Li, and V. Sachdeva, "Bioperf: a benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pp. 163–173, Oct 2005.