Automated Fine-Grained CPU Provisioning for Virtual Machines

DAVIDE B. BARTOLINI, FILIPPO SIRONI, DONATELLA SCIUTO, and
MARCO D. SANTAMBROGIO, Politecnico di Milano

1. INTRODUCTION

Infrastructure as a Service (IaaS) clouds promise to enable business flexibility with a pay-as-you-go model for computation. Within this model, the interest of users is minimizing business costs for executing a given workload with the desired performance, while the interest of vendors is optimizing infrastructure utilization, so as to minimize

Received January 2014; revised April 2014; accepted June 2014

Authors' addresses: D. B. Bartolini, F. Sironi, D. Sciuto, and M. D. Santambrogio, Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Via Ponzio 34/5, 20133 Milano, Italy, {davide.bartolini, filippo.sironi, donatella.sciuto, marco.santambrogio}@polimi.it.

the total cost of ownership (TCO), without breaking service level objectives (SLOs). Current virtualization infrastructures lack tools to easily fulfill these interests: users need to manually determine, for each virtual machine (VM), the amount of resources to rent; providers have to be conservative when consolidating workloads on multicore-powered host nodes to reduce TCO, since colocation can lead to unexpected performance degradation [Farley et al. 2012; Govindan et al. 2011; Mars et al. 2011].

For these reasons, a system able to automatically size and enforce allocations of a contended resource based on user-defined, application-level performance requirements would be a valuable tool to help more easily meet users' and providers' interests. Designing such a system is an interesting and challenging problem. The main contribution of this article is presenting the design and evaluating the benefits of *AutoPro*: a runtime system we developed to serve as such tool.

Recent research [Govindan et al. 2011; Mars et al. 2011; Kasture and Sanchez 2014] on workload consolidation provides techniques to estimate and mitigate performance degradation for consolidated workloads, enabling safe sharing of host nodes. Other contributions [Nathuji et al. 2010; Padala et al. 2009; Shen et al. 2011] describe systems that automate resource allocation based on SLOs but work at a coarse scale and granularity (see Section 5). In contrast, with *AutoPro*, we explore the possibility of enhancing IaaS clouds with automated, fast, and fine-grained resource allocation to meet users' SLOs, while allowing providers to safely share host nodes among VMs to maximize node-level utilization and reduce TCO. Section 2 motivates, through a case study, the advantages that *AutoPro* can bring to IaaS infrastructures.

AutoPro leverages feedback control to automate resource allocation, filling the gap between application-level performance SLOs and allocation of a contended resource, thus dispensing users from the need to explicitly file resource reservations. Moreover, *AutoPro* automatically allocates unclaimed resources to batch workloads on a best-effort basis, enabling providers to maximize node-level utilization. Section 3 discusses our design of *AutoPro* and its subsystems, presenting the methodology that allows *AutoPro* to achieve these goals.

The main strength of *AutoPro* is its ability to take fast decisions and perform precise resource allocations without requiring profiling VMs in advance. This strength comes from the use of a theoretically sound, yet simple, control strategy. *AutoPro* uses controllers based on a resource–performance model that binds VM performance, measured in a metric meaningful to the user, and resource allocation. While the appropriate performance metric (e.g., throughput, latency, turnaround time) depends on the characteristics of the application wrapped in each VM, users do not need to worry about modeling: system developers can determine models that apply to a whole class of applications. Section 4.2 validates a resource–performance model for VMs that expose runtime throughput measurements (e.g., requests/s for a web server) and Section 4.3 evaluates *AutoPro* on managing such VMs. We are extending *AutoPro* to support other performance metrics.

In general, VM performance depends on the supply of different resources (e.g., compute, I/O bandwidth); however, in most cases, a single resource is the bottleneck at any given time. Contention on a single-bottleneck resource will arise in many nontrivial colocation cases, since finding colocation mixes where the performance of different VMs is bound to complementary resources is often not possible. For these reasons, automating allocation of a contended resource to meet performance SLOs is an interesting problem; *AutoPro* copes with it through a resource–performance model that considers the bottleneck resource. To concretely evaluate our approach, this article focuses on throughput-driven, compute-intensive multithreaded applications wrapped inside VMs colocated on a multicore processor, often found in data center host nodes. In this scenario, the compute bandwidth is the bottleneck resource we consider; extending

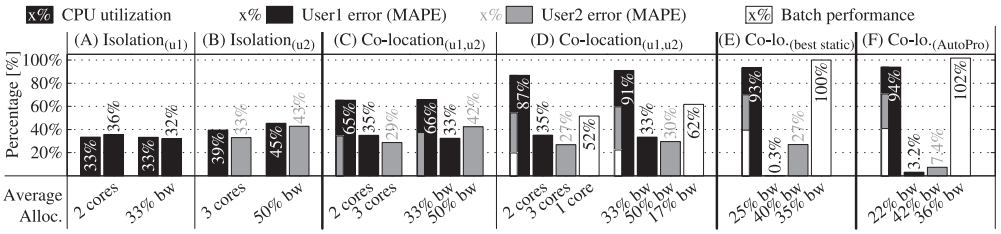

Fig. 1. Host node CPU utilization, mean average percentage error (MAPE) on the performance of VMs by $User_1$ and $User_2$, and performance of the batch VM in different scenarios (A–F). VMs run in isolation (A, B) or are colocated (C–F); we evaluate static allocation of whole cores or CPU bandwidth (bw) (A–E) based on user requests of two and three vCPUs (A–D) or on an oracle (E) and dynamic allocation with *AutoPro* (F). The thin bars in scenarios C–E show the per-VM CPU utilization breakdown, using the same shading as error/performance.

AutoPro to deal with shifting bottleneck resources and different performance metrics is an orthogonal issue we are investigating (see Section 6).

We evaluate *AutoPro* running representative multithreaded applications that stress contention on compute bandwidth. Our extensive experimental evaluation (see Section 4) shows that *AutoPro* meets its goals of attaining SLOs with a lower error than the best static allocation and maximizing node-level utilization.

2. CASE STUDY AND OVERVIEW

In order to provide some context and motivation and to show what advantages *AutoPro* can bring to IaaS infrastructures, we present a case study where two users, u_1 and u_2, deploy a VM to a public IaaS cloud. Each VM wraps a compute-intensive multithreaded application subject to the SLO of maintaining a required performance level, set on an application-specific throughput measurement. In this scenario, users want to achieve the respective SLO with the minimum amount of resources, to minimize business cost, while the cloud provider wants to optimize the data center utilization to minimize the total cost of ownership [Leverich and Kozyrakis 2014].

As a concrete example of this scenario, we colocate two VMs, respectively executing the *swaptions* and *x264* multithreaded applications from the PARSEC 2.1 benchmark suite [Bienia 2011], on a host node with a hexa-core processor (see Section 4.1 for details on its configuration). We set SLOs on application-specific performance metrics (i.e., swaptions/s and frames/s, respectively) such that both VMs can attain their SLOs within the host node capacity. We analyze a set of provisioning scenarios that the service provider might choose; to do so, we leverage the integration of the KVM hypervisor [Kivity et al. 2007] with the Linux kernel's resource containers (control groups, or *cgroups*) [Banga et al. 1999; Menage 2013] to allocate the CPU.[1] The service provider can choose space partitioning or time multiplexing to map the multithreaded applications that run in users' VMs onto the multicore processor of our host node. To evaluate both strategies, we configure all VMs with six virtual CPUs (vCPUs) and allocate the equivalent of n physical CPUs to a VM by either packing its vCPUs onto n cores or by using all cores and capping its CPU bandwidth to $n/6$. Figure 1 reports the performance and node-level CPU utilization of the two VMs in six different provisioning scenarios, marked (A) through (F).[2]

[1]Using *cgroups* for resource partitioning makes all our discussion directly applicable to traditional multiprocessing, where applications are not wrapped in VMs, as *cgroups* support regular Linux tasks.

[2]The results in Figure 1 are the 95th percentile over 30 runs for each scenario.

Scenarios (A) through (D) model current public IaaS clouds, which require users to explicitly rent an integer number of vCPUs. For these scenarios, we pick demands of two and three vCPUs for u_1 and u_2, respectively. These allocations are, for each VM, the integer number of vCPUs that more closely meets the SLO, that is, that minimizes the mean average percentage error (MAPE), our accuracy metric. Intuitively, MAPE gives a measure of how far each VM is, on average, from meeting its performance SLO across its execution; Equation (6), in Section 4.2, reports a formal definition.

Scenario (E) evaluates the best (i.e., with lowest MAPE) static (i.e., fixed throughout VM execution) allocation, which we determined through offline profiling.

Finally, we deploy *AutoPro* on our host node to manage our case study; Scenario (F) reports these results, showing the advantages over current IaaS management policies.

2.1. Analysis of the Case Study

Figure 1(A) and (B) analyzes the baseline scenario in which the provider serves u_1 and u_2 on dedicated CPUs, backing the two and three vCPUs of u_1 and u_2, respectively, with two and three cores (partitioning), or 33% and 50% CPU bandwidth (multiplexing). Using dedicated CPUs (i.e., single-socket host nodes or a multisocket host node) avoids issues due to contention on shared resources (e.g., the last-level cache) but leads to underutilization of the host nodes/sockets (CPU utilization is between 33% and 45%). Low node-level utilization is generally inefficient for the service provider, as it negatively impacts the TCO due to poor energy proportionality of current host nodes [Meisner et al. 2011]. Moreover, the provisioning configuration of this scenario is also inefficient for users, who are far from precisely meeting their SLOs (MAPEs range from 32% to 43%). The performance traces show that both VMs actually outperform the respective SLOs, meaning that users are using, and paying for, more resources than they would really need; this situation is common, as IaaS users tend to overprovision resources [Gmach et al. 2012]. Notice that the vCPUs-to-CPU mapping strategy variably influences CPU utilization and performance for the two VMs. For u_2's VM, which runs *x264*, the MAPE increases more when using six cores at ≈50% CPU bandwidth than when packed on three reserved cores. This higher error is actually due to the VM running faster when using more cores (note that node-level utilization also increases). Instead, u_1's VM, which runs *swaptions*, runs slightly faster when partitioning cores. Section 3.3 elaborates further on partitioning versus multiplexing.

To increase node-level utilization, the provider can colocate the two VMs onto one host node/socket; Figure 1(C) covers this scenario. Colocating VMs improves node-level utilization (reaching ≈65%), as now the two VMs share hardware resources;[3] therefore, VMs' colocation benefits the provider. However, colocation can unpredictably impair the performance of users' VMs due to contention over shared resources (e.g., last-level cache (LLC), memory bandwidth). While this effect is negligible in this specific scenario (MAPEs are very close to the isolation scenarios, and we verify that both VMs still outperform their SLO), because these VMs make for a good colocation pair, more noticeable performance interference arises in other cases, as we will discuss. In general, building good colocation mixes to limit performance interference is a challenging problem that is orthogonal to the scope of this article [Govindan et al. 2011; Mars et al. 2011].

To further increase node-level utilization, the provider can employ the leftover CPU bandwidth to run additional batch applications (e.g., maintenance jobs) on a best-effort basis; Figure 1(D) covers this scenario. To simulate a batch workload, we run

[3]CPU utilization in scenario (C) is less than the sum of the CPU utilizations in scenarios (A) and (B) because, according to data from performance counters, in this case VMs colocation partly hides memory access latencies.

the *psearchy* benchmark [Boyd-Wickizer et al. 2010], set to index the Linux kernel 3.9 source tree, in an additional VM.[4] We provision this batch VM with the compute capacity left unused by the two SLO-bound VMs: either one core or 17% of the CPU bandwidth. Colocating the batch VM bumps node-level utilization up to ≈90% and turns out to reduce the MAPE for u_2's VM. This apparently positive effect is actually the result of a reduction of the performance of u_2's VM, due to the sensitivity of *x264* to contention over the LLC [Cook et al. 2013] caused by cache-intensive corunners, like *psearchy*. Therefore, this effect is not beneficial to u_2, whose VM is still far from matching the SLO and also gets reduced performance while having to pay for the same amount of resources as in the previous scenario.

In all the scenarios we explored so far, users are far from precisely meeting SLOs and they face increased costs due to overprovisioning, since they rent more resources than meeting their SLO would require. We will explore and combine two ways of improvement: (1) automated and (2) finer-grained resource estimates and allocations. If the IaaS cloud could automatically determine the exact resource needs based on the SLOs and allocate resources on a fine-grained scale, the provider could take the burden of estimating resource needs from users. To evaluate the benefits of this scenario, we derive (through offline profiling) that the best static allocations (i.e., the ones that yield the lowest MAPEs on the performance SLO of both user-submitted VMs) are 1.5 vCPUs for u_1 and 2.4 vCPUs for u_2. Since we need to allocate fractions of a core, CPU partitioning is too coarse grained for this case; therefore, we allocate vCPUs by capping the CPU bandwidth. Figure 1(E) shows that, in this case, u_1's VM (i.e., *swaptions*) almost perfectly matches its SLO (with 0.3% error), while the lowest MAPE for u_2's VM (i.e., *x264*) is 27%. The higher error for *x264* is due to this application showing varying execution phases (as Figures 3 and 11 further illustrate); due to this characteristic, no static allocation will be able to closely track the SLO. This scenario is the best we have explored so far, as both users are closer to meeting their SLOs while renting fewer resources, and node-level utilization is kept high; moreover, the CPU bandwidth freed by using just enough resources to minimize the MAPEs goes to increase by ≈60% the performance of the batch workload, benefiting the provider.

With *AutoPro*, we show that it is even possible to improve over Scenario (E) by leveraging a resource–performance model, runtime performance measurements, and dynamic fine-grained resource allocation. Figure 1(F) reports the results we obtain when deploying *AutoPro* to manage our case study: both VMs closely match the respective SLO (MAPEs are 3.2% and 7.4%, respectively) and both node-level utilization and performance of the batch VM are maximized. *AutoPro* reduces the error for *x264* with respect to Scenario (E) thanks to a fast dynamic allocation that, in contrast to static allocation, can react to varying execution phases; moreover, *AutoPro* keeps the error for *swaptions* low. Section 4.3.3 picks one of the runs of this scenario and provides further insights. Section 3 explains how we designed and implemented *AutoPro* to achieve such results.

Notice that, while the performance interference between u_2's and the batch VM impairs the colocation efficiency, *AutoPro* is robust to contention on unmanaged shared resources (as we discuss in Section 4.3.2). By exploiting performance feedback, *AutoPro* can realize a soft partitioning of unmanaged resources by adjusting the allocation of the bottleneck resource [Fedorova et al. 2007; Sironi et al. 2012]. Nonetheless, limiting the contention on unmanaged resources is still valuable, since this way of attacking degradation is inefficient and can complicate the billing schema (see Section 6).

[4]Similarly, the SPECvirt_sc2013 benchmark [Standard Performance Evaluation Corporation 2013] periodically runs file compression tasks in an otherwise idle VM to simulate batch workloads.

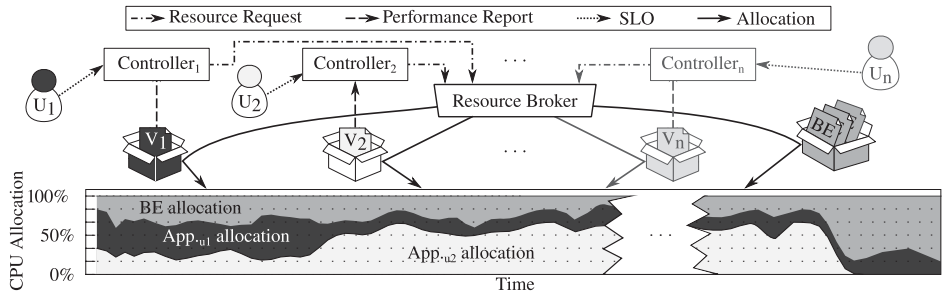

Fig. 2. *AutoPro* uses per-VM (V_i) controllers to estimate the resource needs to meet SLOs defined by users (U_i). A resource broker aggregates demands and determines allocations, fair-sharing leftover capacity among batch best-effort (BE) VMs.

3. AUTOMATED FINE-GRAINED PROVISIONING

The idea at the base of *AutoPro* is extending IaaS clouds with fine-grained automated resource allocation in order to provide users with a simple interface for specifying SLOs on understandable performance metrics. Figure 2 depicts the architecture of *AutoPro*. A resource container enforces allocations to each VM; a dedicated controller estimates, based on performance reports, the amount of resources each SLO-bound VM needs to meet its SLO; and a broker collects resource demands and determines allocations, giving leftover resources to an additional resource container that fairly provisions batch VMs on a best-effort basis. The controllers that estimate the resource need for each VM leverage an adaptive resource–performance model that captures the relationship between performance measurements and allocation of the contended resource; Section 3.2 elaborates on the defined *AutoPro*'s control schema.

AutoPro deals with resource allocation and not with admission control or workload placement, which are related but orthogonal issues. We rely on the presence of an admission control system that chooses VM placement across the data center [Chen et al. 2012; Schwarzkopf et al. 2013], and we assume that the hypervisor supports live migration [Jo et al. 2013] in case a node becomes overloaded. While both VM placement and migration present open questions for research, *AutoPro* focuses on a different challenging and interesting problem: automating allocation of a contended resource within a single node, based on application-level performance SLOs.

The innovative contribution of this article is showing how it is possible to extend IaaS clouds toward a *Performance-as-a-Service* model [Bartolini et al. 2013b] that benefits both users, with SLO-based automated provisioning, and providers, with support for system-level management of idle resources. Thanks to fast and precise allocation decisions, *AutoPro* can adapt to time-varying workloads, automating the allocation of a contended resource to meet SLOs and maximize node-level utilization.

3.1. Performance Metrics and Measurements

AutoPro requires each SLO-bound VM to make periodic performance reports available to its controller, in order to leverage its resource–performance models, as proposed in previous works [Zhang et al. 2002; Padala et al. 2009; Shen et al. 2011; Sironi et al. 2012; Bartolini et al. 2013a; Hoffmann et al. 2013; Sironi et al. 2014]. Any performance metric meaningful to the user can be used for these reports and to express SLOs; for instance, a web server can use throughput (e.g., requests/s for a web server) or latency (i.e., response time).

While this article focuses on throughput as the reference performance metric, *AutoPro* can be extended to support other performance metrics, provided that the appropriate resource–performance models are made available. Notice that application

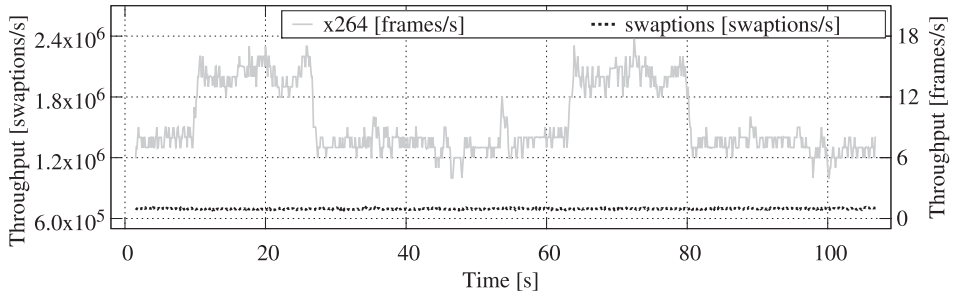

Fig. 3. Throughput of six-threaded *swaptions* [swaptions/s] and *x264* [frames/s], colocated on our hexa-core host node with static CPU bandwidth allocations.

developers do not have to worry about resource–performance models: they simply need to provide performance measurements according to one or more supported performance metrics.

To make sure we evaluate the design and implementation of *AutoPro* on a varied set of multithreaded workloads, we use several applications from the PARSEC 2.1 benchmark suite [Bienia 2011] (see Section 4). While these benchmarks were not designed to capture all the characteristics of typical cloud workloads, colocating PARSEC applications does create contention on compute bandwidth, thus stressing the problem that this article addresses. Since PARSEC applications do not natively report performance at runtime, we instrument a subset of the suite[5] to report throughput through our efficient user-space implementation of the Application Heartbeats API [Hoffmann et al. 2010; Sironi et al. 2012; Bartolini et al. 2013a; Sironi et al. 2014]. The hypervisor accesses VM performance measurements as in previous work [Padala et al. 2009; Shen et al. 2011]. In real deployments, performance reports may be obtained from application logs or monitoring infrastructures.

The controllers we use to determine resource allocations (see Section 3.2.1) work on throughput measurements computed as a moving average on a sliding window. To give an example of how this metric can characterize the performance of PARSEC applications, Figure 3 shows the traces of two VMs with radically different behavior: with a constant resource allocation, *swaptions* has a stable performance throughout its execution, while *x264* shows input-dependent[6] oscillations [Sasaki et al. 2012; Bartolini et al. 2013a; Sironi et al. 2014]. Measuring throughput over an adequate sliding window allows the controllers to promptly catch transitions between execution phases and adjust resource allocations as needed (see Section 4.3).

3.2. Estimating Resource Needs

AutoPro periodically evaluates throughput measurements against the SLO and updates resource allocations to keep the performance of each VM in line with its SLO. We devise a control schema that splits this process into two levels:

(1) estimating resource demands independently for each SLO-bound VM and
(2) aggregating demands and determining allocations.

Figure 4 illustrates this schema, focusing on one VM. At each control step, (1) a dedicated *VM Controller* files a resource request, determined through an adaptive proportional-integral (PI) controller [Åström and Wittenmark 2008]; and (2) a resource

[5]We pick applications that allow a simple and meaningful definition of throughput measurements.
[6]Here we encode the PARSEC native input twice.

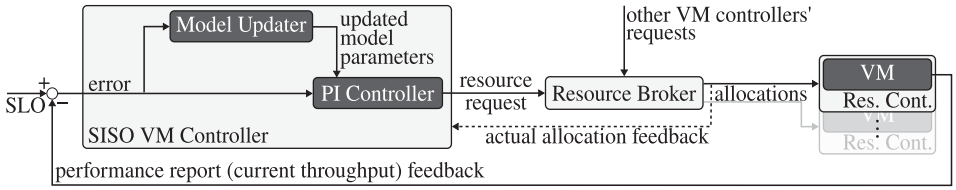

Fig. 4. Overall architecture of *AutoPro*'s control schema.

broker (RB) aggregates these requests, determines the actual allocations based on system-level policies, and enforces them through the resource containers.

3.2.1. VM Controllers. *VM Controllers* (see the left-hand part of Figure 4) take care of continuously estimating the CPU bandwidth each VM needs to match its SLO.

The core of the *VM Controller* is a resource–performance model that binds the VM performance over a given time window (i.e., its throughput over that time window) to resource allocation. Equation (1) formalizes this resource–performance model, where $t_w(k)$ and $t_w(k+1)$ are the average throughputs measured at control steps k and $(k+1)$ on a window of w seconds, respectively, and $r(k)$ is the CPU bandwidth the VM is provisioned with during the time quantum $(k, k+1)$:

$$t_w(k+1) = a \cdot t_w(k) + b \cdot r(k). \tag{1}$$

The terms a and b are parameters that characterize the workload and the scalability of the VM; for the model to be accurate, these two parameters must adapt to the characteristics of each VM. We use a Recursive Least Squares (RLS) filter (see the *Model Updater* in Figure 4) to continuously update the a and b parameters based on observed throughput and allocated resources. Through this adaptive approach [Åström and Wittenmark 2008], we avoid the need for profiling VMs in advance and support VMs with varying load. Section 4.2 evaluates the accuracy of this model and of the online estimation.

In order to estimate resource requests based on the model in Equation (1), we leverage formal control theory to synthesize a PI controller: we derive an expression of the resource request $r(k)$ at step k as a function of the resource assignment $r(k-1)$ granted during the quantum $(k-1, k)$, the performance error $e(k) = \bar{t} - t_w(k)$, and the parameters a and b. First, in Equation (2), we determine the transfer function $\mathcal{P}(z)$ by applying the $\mathcal{Z}$-transform to Equation (1):

$$\begin{aligned} z \cdot \mathcal{T}_w(z) &= a \cdot \mathcal{T}_w(z) + b \cdot \mathcal{R}(z) \\ \mathcal{P}(z) = \frac{\mathcal{T}_w(z)}{\mathcal{R}(z)} &= \frac{b}{z-a}. \end{aligned} \tag{2}$$

$\mathcal{T}_w(z)$ and $\mathcal{R}(z)$ are the $\mathcal{Z}$-transforms of the throughput measurement $t_w(k)$ and VM-owned resources $r(k)$, respectively.

We leverage formal control theory to synthesize the PI controller by constraining the transfer function of the feedback loop [Hellerstein et al. 2004]; Equation (3) shows this operation:

$$\mathcal{L}(z) = \frac{\mathcal{C}(z)\mathcal{P}(z)}{1 + \mathcal{C}(z)\mathcal{P}(z)} \triangleq \frac{1-p}{z-p}, \quad p \in (0, 1). \tag{3}$$

$\mathcal{L}(z)$ and $\mathcal{C}(z)$ indicate the transfer functions of the feedback loop and the PI controller, respectively. We set $\mathcal{L}(z)$ to a first-order transfer function with one pole in p. The constraint $p \in (-1, 1)$ ensures that the closed loop is asymptotically stable; furthermore, $p \in (0, 1)$ guarantees convergence without overshooting and oscillations [Hellerstein

et al. 2004]. Within this interval, the parameter allows one to set a tradeoff between fast response and safety: smaller values of p will result in faster responses. We find that the actual value of p in the $(0, 1)$ range does not have considerable impact on the behavior of our controllers; we ascribe this fact to the nonlinearities that our linear resource–performance model discards. We empirically find $p = 0.1$ to be a good choice and use this value in all our experiments.

To actually find an expression for $r(k)$, we start by deriving, in Equation (4), an expression of $\mathcal{C}(z)$ by combining Equations (2) and (3); we impose the transfer function of the PI controller to be the ratio between VM-owned resources $\mathcal{R}(z)$ and performance error $\mathcal{E}(z)$:

$$\mathcal{C}(z) = \frac{(1-p)\cdot(z-a)}{b\cdot(z-1)} \triangleq \frac{\mathcal{R}(z)}{\mathcal{E}(z)}. \tag{4}$$

Elaborating further and applying the $\mathcal{Z}$-antitransform and a time shift, we finally get, in Equation (5), the desired expression for $r(k)$:

$$r(k) = r(k-1) + \frac{1-p}{b}\cdot e(k) - a\cdot\frac{1-p}{b}\cdot e(k-1). \tag{5}$$

The PI controller in each *VM Controller* uses Equation (5) to estimate the resource request that should let the VM match its SLO in the next control period.

We implemented the *VM Controller* in C++, using the *Armadillo* library [Sanderson 2010]. The simplicity of our model allows a fast implementation, which enables *AutoPro* to run with an aggressive control period to react to quick workload changes: our implementation takes less than $10\mu s$ to update the parameter models and compute the next allocations in all our experiments on our reference machine (see Section 4.1 for details on its configuration). This very small overhead allows us to use faster control periods (see Section 4.1) than previous works [Padala et al. 2009; Shen et al. 2011], achieving finer-grained control (as we discuss in Section 5).

3.2.2. Controllers Stability. The stability theorem of formal control theory for linear, time-invariant systems states that a system represented by a transfer function $\mathcal{G}(z)$ is stable if and only if the poles (i.e., the roots of the dominator polynomial) of $\mathcal{G}(z)$ are within the unit circle in the complex coordinate plane [Hellerstein et al. 2004]. Our choice of $p = 0.1$ provides such guarantee, since $z = p$ is the only pole of the feedback loop transfer function, as Equation (3) shows.

The pole-elision method we use to synthesize our controllers adds the requirement that the elided poles (in our case, the only pole a, as Equation (2) shows) must not be unstable. To abide to this condition, we verify that our model updater always returns parameter estimates such that $|a| \in (0, 1)$.

Whenever VMs expose a steady behavior (i.e., within a phase), the RLS filter we use as our model updater will make the parameters a and b converge to their "real" values and, thanks to the stability properties we just discussed, the VM controllers will drive performance toward SLOs. When the workload of a VM varies over time, switching its resource–performance behavior (i.e., going through different phases), the model updater will take some control steps to converge to the new parameter values, causing transient oscillations of the allocation on phase boundaries.

3.2.3. Resource Brokerage and Allocation. At each control step, the RB collects all the resource requests from the *VM Controllers* and determines the actual allocations. This aggregation step enforces global constraints, such as CPU capacity, and applies global policies (e.g., how to use unclaimed resources).

We consider two constraints on allocations: a global maximum (max) and a per-VM minimum (min). The max bound enforces CPU maximum capacity, while the min bound

avoids allocating too-small CPU fractions, which can cause imprecise enforcement at the resource container level.

A resource request sum exceeding the *max* bound indicates that the CPU capacity is not enough to let all SLO-bound VMs deliver the respective required throughput. If this case occurs, the RB can make one of two decisions:

(1) migrate one or more VMs to a less loaded node or
(2) fairly distribute the available capacity to VMs proportionally to their original requests.

While migrating VMs might be necessary to respond to chronic resource shortage, this operation has nonnegligible cost and should be avoided when resource scarcity is due to short heavier phases or control glitches because, in these cases, proportionally distributing the available capacity would at most cause a transient drop in performance. In this article, we only evaluate cases where resource scarcity happens only temporarily and do not consider VM migration, which represents an extreme way out to make up for poor VM placements by the admission control system.

When the sum of resource requests is below the *max* bound, the RB chooses how to use the unclaimed resources. Since this article considers the objective of maximizing node-level utilization, *AutoPro* employs unclaimed resources to execute additional batch VMs on a best-effort basis.

In general, the RB could implement other policies to pursue different objectives, for instance, idling unused resources and triggering low-power states to reduce power draw, or redistributing unclaimed resources to SLO-bound VMs, based on a priority schema, to support different QoS levels [Nathuji et al. 2010].

3.3. Containers and Resource Provisioning

To enforce allocations, *AutoPro* relies on the ability of the hypervisor to partition resources and allocate them to VMs through a resource container mechanism [Banga et al. 1999]. We base our implementation on the Kernel Virtual Machine (kvm) hypervisor [Kivity et al. 2007], which uses Linux *control groups* (*cgroups*) [Menage 2013] for resource partitioning. The vCPUs of a VM appear as processes to the hypervisor, which groups them into the same *cgroup*, and this allows one to set caps on resource usage. We exploit *cgroup* hierarchies to create an additional *cgroup* that contains all the batch VMs, which fairly share the resources unclaimed by SLO-bound VMs.

Linux *cgroups* offer a few *subsystems* to allocate different types of resources (e.g., cores, CPU bandwidth, block I/O bandwidth). Since this article focuses on managing VMs that run compute-intensive multithreaded tasks, we use the *cpuset* and *cpu* [Turner et al. 2010] subsystems to allocate fractions of the CPU. These two subsystems control CPU partitioning and multiplexing: the *cpuset* subsystem allows one to pin threads (in our case, vCPUs) in a *cgroup* to a subset of the available cores, while the *cpu* subsystem allows one to set a cap on the CPU bandwidth each *cgroup* can use [Turner et al. 2010].[7]

AutoPro can support both CPU partitioning and multiplexing; we evaluated both mechanisms on managing VMs executing multithreaded applications from the PARSEC 2.1 benchmark suite colocated on our hexa-core host node (configuration details are in Section 4). We verify that these applications scale well to six threads [Sasaki et al. 2012] and configure VMs with six vCPUs. To allocate the host CPU, we either pack vCPUs on a subset of the host cores or cap the host CPU bandwidth [Turner et al. 2010].

[7]The Xen hypervisor [Barham et al. 2003] offers similar facilities [Xen Project 2013].

Applications with poorer scalability or host nodes with a higher core count might lead to scalability issues, which were tackled in recent research [Sasaki et al. 2012]. While these issues would impair the efficiency of our system, this problem is orthogonal to the scope of this article; Section 6 sketches a solution based on partitioning the multicore in smaller time-multiplexed islands.

We found that *AutoPro* obtains similar results, in terms of accuracy in enforcing SLOs, utilization, and batch VM performance, with both allocation mechanisms (i.e., dynamic partitioning or multiplexing). While the overall results are similar, CPU multiplexing allows more stable allocations, thanks to its finer granularity (partitioning forces minimum allocation granularity to one core) and we base our evaluation (Section 4) on this mechanism.

4. EVALUATION

We evaluate *AutoPro* in two steps: Section 4.2 validates our resource–performance model against both single- and dual-VM workloads; Section 4.3 evaluates *AutoPro* on managing workloads made of two SLO-bound VMs and a batch VM.

4.1. Platform, System, and Applications

Our host node is a Dell Precision T3500 workstation with an Intel Xeon W3690 hexacore processor, clocked at 3.46GHz and equipped with 12MB of shared LLC, and 12GB of main memory, distributed on three channels and clocked at 1066MHz. We disable the Enhanced Intel SpeedStep and Turbo Boost Technologies to bind the processor frequency to its nominal value. Due to the lack of cache partitioning mechanisms, we cannot directly address (i.e., eliminate) conflicts on the shared LLC; in our evaluation, we show that *AutoPro* is robust to performance interference due to this limitation. We disable the Intel Hyper-Threading Technology (i.e., simultaneous multithreading), which could give performance benefits for some workloads but would introduce contention on core-private caches, thus possibly exacerbating the performance interference problem, which we already show *AutoPro* can cope with.

We use Debian GNU/Linux 7.0 with kernel version 3.9 and configure the *cpu* subsystem of *cgroups* to enforce CPU bandwidth caps on a period of 50ms. We configure *AutoPro* to sample VM throughput on a sliding window of 750ms and update the resource–performance models every 50ms, and allocate resources every 750ms. We empirically determine these parameters to yield precise resource allocation and responsiveness on our benchmarks: shorter periods lead to imprecise resource allocations with the *cpu* subsystem of *cgroups*, while longer periods lead to unresponsiveness to execution phases.

To evaluate *AutoPro* on a varied set of multithreaded applications, we use *blackscholes*, *bodytrack*, *canneal*, *dedup*, *ferret*, *swaptions*, and *x264*, from the PARSEC 2.1 benchmark suite [Bienia 2011], as our workloads for SLO-bound VMs. We configure each VM with six vCPUs and each application to run with six threads except for *dedup* and *ferret*, which use six threads per pipeline stage. We synchronize the execution of the regions of interest through the *hooks* provided by the benchmark suite. We use the *native* inputs for all applications except for *swaptions*, which evaluates 96 swap options, and *x264*, which encodes its native input video twice.

We use the indexing part of the *psearchy* (i.e., *pedsort*) benchmark [Boyd-Wickizer et al. 2010] to simulate a batch workload not bound to a SLO. We use *psearchy* to index the Linux kernel 3.9 source tree using six threads, each using at most 1GB for its hash table. Just as for the PARSEC applications (see Section 3.1), we instrumented *psearchy* to report throughput defined, in this case, as the number of jobs per hour per thread.

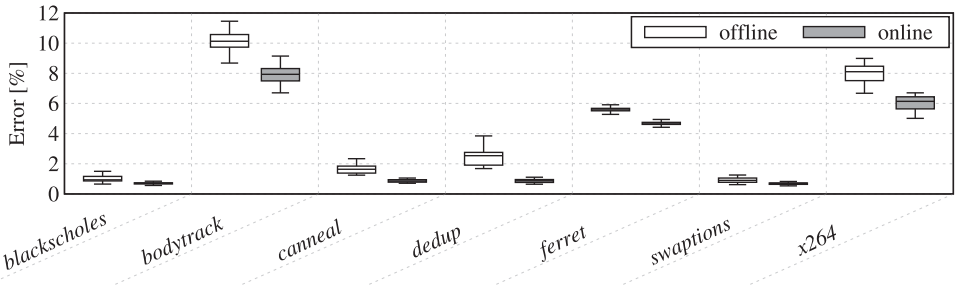

Fig. 5. MAPEs on solo runs, using offline (with the least squares algorithm) or online (with the *Model Updater*) estimation. Whiskers depict the minimum and maximum errors; boxes depict 25th, 50th, and 75th percentiles.

4.2. Resource–Performance Model Evaluation

We validate the accuracy of our resource–performance model, defined in Equation (1), which binds resource allocations $r(k)$ (i.e., a percentage of the CPU bandwidth) to VM throughput $t_w(k)$ (i.e., moving averages on a sliding window w of 750ms). Each VM runs one of the PARSEC applications we selected and we evaluate both solo runs (single VM) and workloads with two colocated VMs (dual VM).

At each control step k (i.e., every 750ms), we randomly vary the resource allocation $r(k)$ of each VM and we log the current throughput $t_w(k)$ and the corresponding resource allocation $r(k-1)$; for the dual-VM workloads, we randomly partition the host CPU capacity among the two VMs. We execute each workload 30 times to achieve statistical relevance.

We use the mean absolute percentage error (MAPE) metric, computed as in Equation (6), to evaluate the model accuracy:

$$\text{MAPE} = \frac{1}{K}\sum_{k=0}^{K}\left|\frac{\widehat{t_w}(k) - t_w(k)}{t_w(k)}\right|. \tag{6}$$

K is the number of control steps, $t_w(k)$ is the actual throughput sampled at step k, and $\widehat{t_w}(k)$ is the throughput estimate according to our model, defined in Equation (1).

AutoPro uses a *Model Updater* (see Section 3.2.1) to compute the model parameters a and b online. Besides evaluating the accuracy of our model through the MAPE metric, we also validate this filter by comparing it against the optimal offline parameters estimated with the least squares algorithm.

4.2.1. Prediction Error on Single-VM Workloads. To evaluate our model in the absence of contention on shared resources, we first consider solo runs of VMs executing each of the PARSEC applications. Figure 5 reports the MAPE metric for our model when using both online and offline parameter estimates. These results show a low prediction error (within 5%) for five out of seven applications and indicate that updating the model parameters online with the *Model Updater* improves accuracy for all the workloads compared to offline estimates of the optimal static parameters. Our results compare favorably with previous work leveraging a similar resource–performance model [Sharifi et al. 2011] (see Section 5).

The prediction error on the solo runs using online estimation is close to 1% for four out of seven workloads; we found that the higher error on *bodytrack* (≈8%), *ferret* (≈5%), and *x264* (≈6%) is due to specific characteristics of these applications; the remainder of this section reports our observations.

The only meaningful throughput metric for *bodytrack* (i.e., frames/s) leads to low measurements: 4 frames/s on average with peaks of 8 frames/s with the highest resource allocation. These low measurements imply that *bodytrack* only completes 3 to 6 frames in the 750ms sliding window we measure throughput on; this dynamic leads to sizable variations (up to 33% on average) due to sampling even when the actual throughput varies only slightly. This jitter impairs the accuracy of our model. While using a longer sliding window and allocation period improves accuracy on *bodytrack*, it would reduce responsiveness on applications with varying phases. We found that 750ms is a good tradeoff to support all our benchmarks; application-specific optimizations could further improve accuracy.

x264 shares the low-throughput issue with *bodytrack*, albeit with lesser impact (measurements can be as low as 6 frames/s). Moreover, it shows input-dependent execution phases (see Figure 3), which introduce variations even with a stable resource allocation. For this reason, the *Model Updater* needs to adapt to the changing workload, by updating the model parameters, and its prediction error increases during the transition to a different execution phase. We could tune the *forgetting factor* of the *Model Updater* to address this issue; however, a more aggressive forgetting factor would make the *Model Updater* more subject to occasional noise.

ferret exploits pipeline parallelism and runs with 24 threads: six threads for each of the four pipeline stages. This is the result of a questionable design: the application is simple from a developer standpoint; however, it can also be highly inefficient due to load imbalance and I/O bottlenecks [Navarro et al. 2009]. Redesigning the applications following Navarro et al. [2009] suggestions would greatly improve the predictability of the application and reduce the prediction error of our resource–performance model.

4.2.2. Prediction Error on Multi-VM Workloads. Since *AutoPro* aims at supporting workload consolidation to maximize node-level utilization, we evaluate the accuracy of our model on dual-VM workloads. Figure 6 reports that the prediction error remains low, indicating that our model is robust against performance degradation due to contention on unmanaged resources. These results follow very similar trends to the single-VM workloads (compare to Figure 5): the maximum error is below 1% for three applications (*blackscholes*, *canneal*, *swaptions*) and overall below 12%, when the *Model Updater* is in place. The same considerations of Section 4.2.1 hold, to a greater extent as a result of the contention on unmanaged resources, for the higher error observed with *bodytrack*, *ferret*, and *x264*.

dedup and *x264* are the two applications whose values of MAPE increase the most when shifting from isolation to consolidation; we ascribe this behavior to the highest sensitivity to the contention on unmanaged resources [Cook et al. 2013].

The low values of the MAPE metric for both single- and multi-VM workloads indicate that our resource–performance model accurately predicts the behavior of the compute-intensive multithreaded applications we employed in these experiments. In addition, the *Model Updater* improves the model accuracy over offline estimates, besides dispensing from the need of profiling applications. The *Model Updater* also makes the model more resilient to the negative effects resulting from the contention on unmanaged resources (e.g., see *x264* in Figure 6).

4.3. AutoPro Evaluation

To evaluate *AutoPro*, we use a strategy similar to the validation of the aresource–performance model. First, Section 4.3.1 shows that *AutoPro* can automatically allocate the right amount of CPU bandwidth to SLO-bound VMs running solo on our host node (see Section 4.1 for configuration details). Second, Section 4.3.2 shows that *AutoPro* can both automate CPU provisioning and maximize node-level utilization when managing

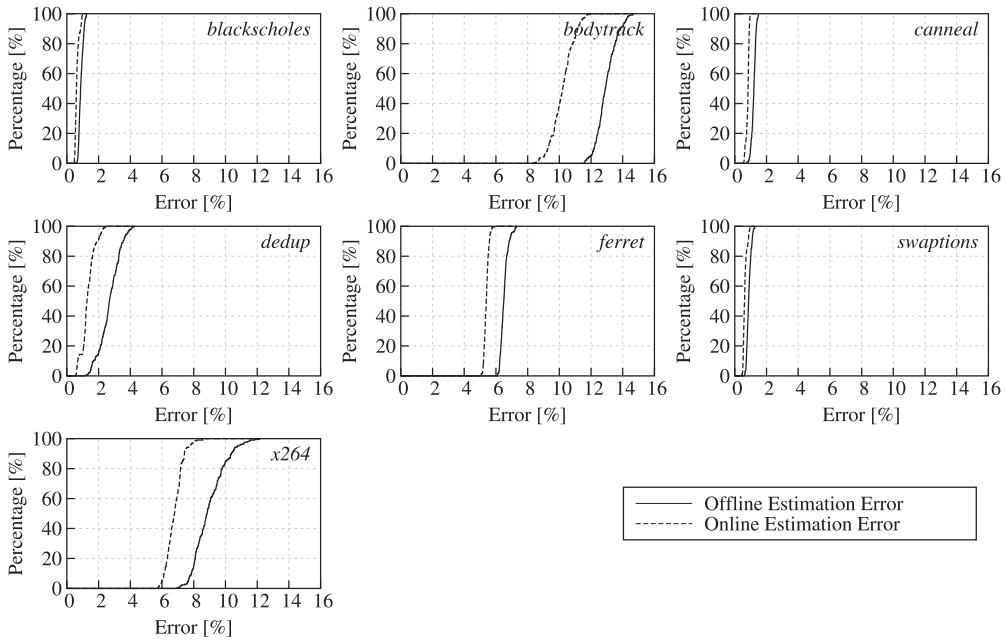

Fig. 6. Each plot shows the cumulative distribution function (CDF) of the MAPE for the VM in the label, considering all workloads where it is present (i.e., 7×30). We report results using both offline (least squares) and online (*Model Updater*) estimation.

two SLO-bound VMs colocated with a batch VM on our host node. In addition, Section 4.3.3 analyzes the dynamic behavior of *AutoPro* in one of the experiments of the motivational case study. The results of these experiments show the following:

(1) *AutoPro* is a practical tool for automating the allocation of a contended resource (specifically, CPU bandwidth) in a data center host node based on performance requirements.
(2) *AutoPro* allows the maximization of node-level utilization and it is robust to the performance degradation resulting from the contention on unmanaged resources.

Each SLO-bound VM executes one of the PARSEC applications we selected, while the batch VM executes *psearchy* (see Section 4.1). We set SLOs as a fraction of the average throughput each application achieves in a solo run on our host without any caps on resource usage (from now on, we refer to this value as the application's *reference throughput*).

To evaluate the ability of *AutoPro* to enforce SLOs, we use the same metric we used in our model evaluation (i.e., the MAPE). Referring to Equation (6), in this case $\widehat{t_w}(k)$ is the actual throughput measurement and $t_w(k)$ is the desired throughput (i.e., the SLO).

To evaluate the ability of *AutoPro* to maximize node-level utilization, we consider how close the average host node CPU utilization during each experiment is to 100%.

4.3.1. Managing Single-VM Workloads. We evaluate the effectiveness of *AutoPro* in allocating just enough resources to single-VM workloads bound to an SLO ranging from 30% to 90% of each PARSEC application's reference throughput, with steps of 20%; Figure 7 shows these results. Most of the errors are (often considerably) below 5%; we analyze the cases of higher error.

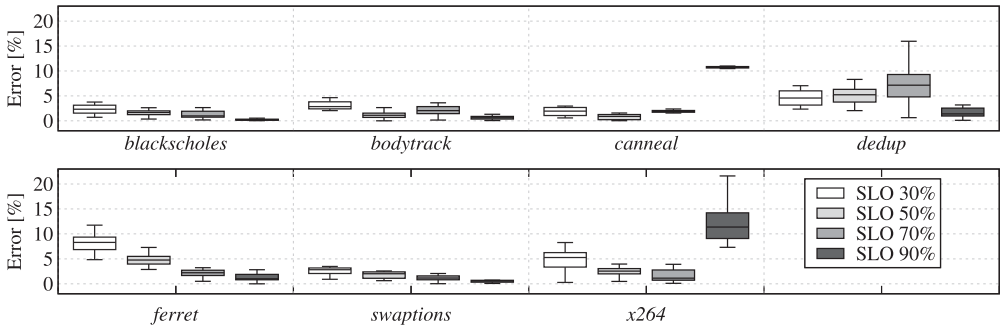

Fig. 7. MAPEs on SLO enforcement for single-VM runs. SLOs range from 30% to 90% of each application's peak performance. Whiskers indicate the minimum and maximum errors; boxes indicate 25th, 50th, and 75th percentiles.

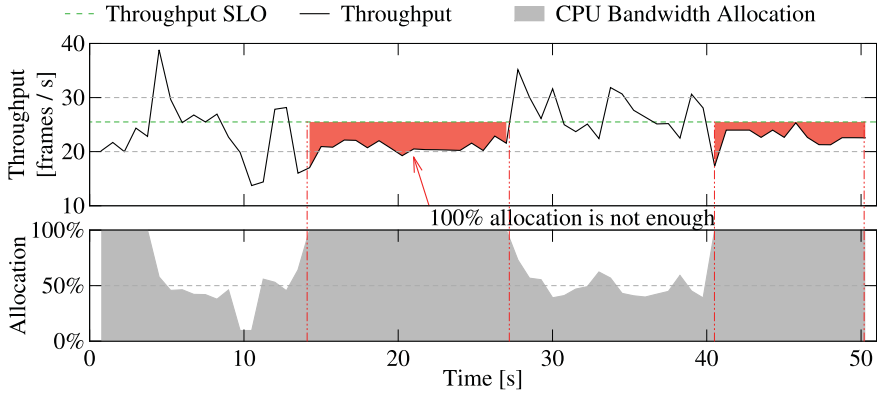

Fig. 8. Excerpt from a run of *x264* bound to a SLO of 90% its average solo throughput. Allocating 100% CPU bandwidth is still not enough to match the SLO during heavier phases.

The results for *x264*, *dedup*, and *ferret* present a sensibly higher variance; in these cases, the same application-specific considerations reported in Sections 4.2.1, 4.2.2 hold.

Different considerations hold for the higher errors of *canneal* and *x264* when setting the SLO to 90% of the respective reference throughput. These applications present considerable throughput variations, even with a fixed resource allocation, due to changes in their workload. *canneal* implements simulated annealing and, due to the structure of this algorithm, it performs more frequent exchange operations at the beginning of its execution (i.e., at "high temperature"), thus showing a constantly decreasing throughput, which is measured in exchanges/s. *x264* presents input-dependent execution phases triggered by the different structure of the frames in the incoming video stream [Sasaki et al. 2012; Bartolini et al. 2013a; Sironi et al. 2014]. For this reason, these two applications present heavier execution phases for which they cannot sustain 90% of their reference throughput across the whole execution, even with all the available resources; Figure 8 exemplifies this issue for *x264*. The red-shaded areas highlight heavier execution phases during which the VM does not achieve the SLO even though *AutoPro* consistently allocates 100% CPU bandwidth. This analysis shows that, in these cases, *AutoPro* still works properly; simply, the SLO is not attainable during some execution phases on our host server. In this situation, the provider might simply terminate the VM and report execution failure due to wrong settings or migrate the VM

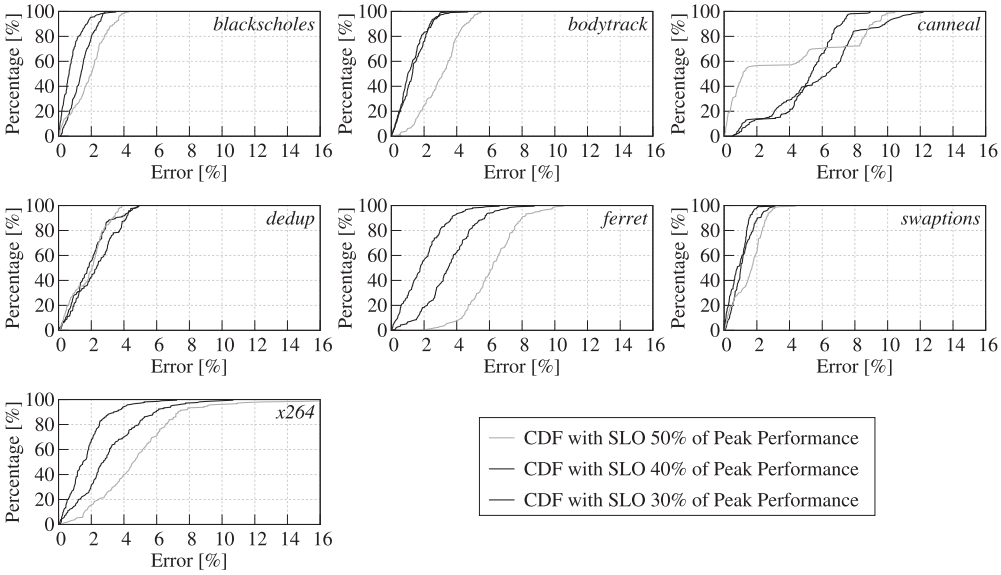

Fig. 9. Each plot shows the CDF of the MAPE for the VM in the label, considering all workloads where it is present (i.e., 7×30). In each experiment, one VM has an SLO of $x = \{30, 40, 50\}\%$ its reference throughput, as indicated in the legend, while the colocated VM has an SLO of $(80 - x)\%$ its reference throughput. All workloads make up a batch VM treated in a best-effort fashion.

to a more powerful node. While this article does not directly deal with VM migration, we are working on a smart VM migration system able to tell chronic from transient resource shortage and, in the latter case, decide to migrate the VM.

4.3.2. Managing Multi-VM Workloads. To evaluate *AutoPro* in a more comprehensive scenario, we use it to allocate resources to two SLO-bound VMs, running PARSEC applications, while maximizing node-level utilization by provisioning unused CPU bandwidth to a batch VM, running *psearchy*, on a best-effort basis. We run every possible colocation of the seven PARSEC applications, repeating each experiment 30 times, and, for each VM, we analyze all the runs that compose it, using the MAPE metric to evaluate the ability of *AutoPro* to enforce SLOs and evaluating the average host node CPU utilization. For each experiment, we set the SLO of one VM to $x = \{30, 40, 50\}\%$ its reference throughput and the SLO of the colocated VM to $(80 - x)\%$ its reference throughput. Given these SLOs, there is no guarantee that the host node capacity will be enough to provision both VMs with the right amount of CPU bandwidth. Hence, some of the workloads may never have values of MAPEs equal to 0, even if the prediction error for those specific workloads is 0.

Figure 9 reports, on these experiments, the distributions of the MAPE for each application executed in an SLO-bound VM. These results show that *AutoPro* meets the three SLO levels with a small error margin (always within 5% and lower in many cases) for four of the seven PARSEC applications (namely, *blackscholes*, *bodytrack*, *dedup*, and *swaptions*); in this respect, *AutoPro* is competitive with SLO-violation results reported in prior work [Shen et al. 2011] (see Section 5 for additional considerations).

We ascribe the higher (but still below 10% at the 90th percentile in all cases) error reported for *ferret*, *canneal*, and *x264* to the same issues we discussed earlier in this section. In addition, for what regards *canneal* and *x264*, the effect illustrated in Figure 8 has sizable effects at SLO levels lower than 90%. The reason is that now there are two colocated SLO-bound VMs that contend on the available host node CPU

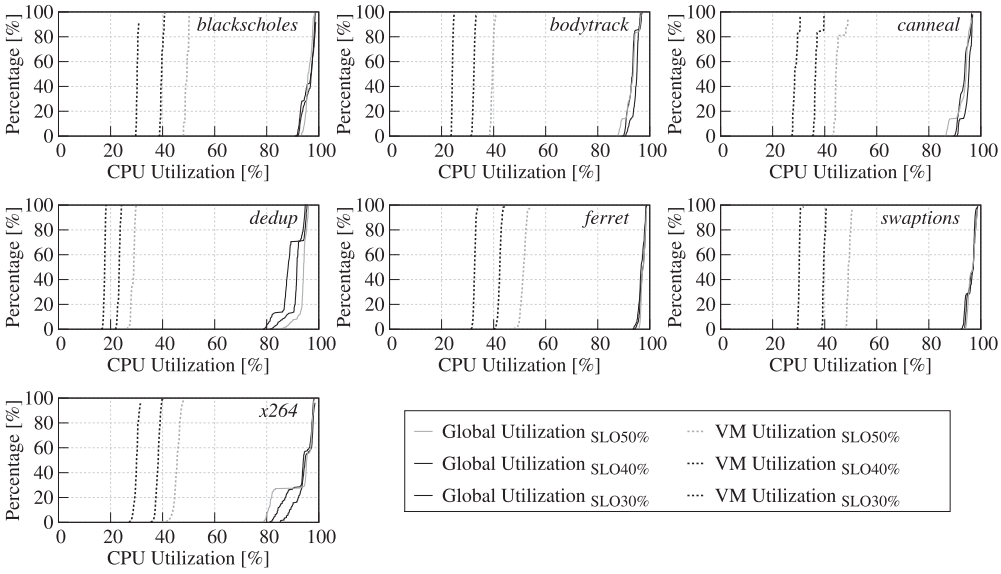

Fig. 10. Each plot shows the CDF of the global and per-VM host node CPU utilization registered in the experiments reported in Figure 9.

capacity, leading to a higher possibility that heavier execution phases lead to resource scarcity. The same possible solution to this issue discussed for the single-VM case (see Section 4.3.1) applies here.

Figure 10 reports the CDF of both global and per-VM host node CPU utilization for all the experiments with three colocated VMs (i.e., two SLO bound and one batch). The plots show that the utilization of the host node CPU is always close to 100% in all cases for all SLO levels. This result confirms that *AutoPro* can effectively maximize node-level utilization by allocating unused resources to batch workloads on a best-effort basis. The per-VM utilization CDFs show that *AutoPro* enacts consistent allocations across the different experiments. The variations in these values are due to adjustments *AutoPro* makes to respect SLOs despite the workload-dependent negative effects of contention on unmanaged resources.

4.3.3. Dynamic Behavior. We conclude our evaluation by analyzing more in detail Scenarios (E) and (F) of our case study (see Section 2.1 and Figure 1). Figure 11 picks the 95th percentile execution of the multi-VM workload composed by *swaptions*, *x264*, and *psearchy* and shows how VM performance and CPU bandwidth allocations vary throughout the experiment. *AutoPro* achieves near-optimal allocation for the VM running *swaptions*, which has stable performance, and tracks much better (with respect to the best static allocation) the SLO for the VM running *x264* by provisioning (at high frequency) the CPU bandwidth allocation. Notice that allocating 100% of the CPU bandwidth does not imply a host node CPU utilization of 100% due to practical issues (e.g., trapping, context switching, synchronization).

5. RELATED WORK

The problem with building virtualization infrastructures able to support auto-scaling of applications and maximize node-level utilization is the natural evolution following the rise of public IaaS and Platform as a Service (PaaS) clouds. This article addresses this problem by starting with a clean slate, “reversing” the classic resource allocation

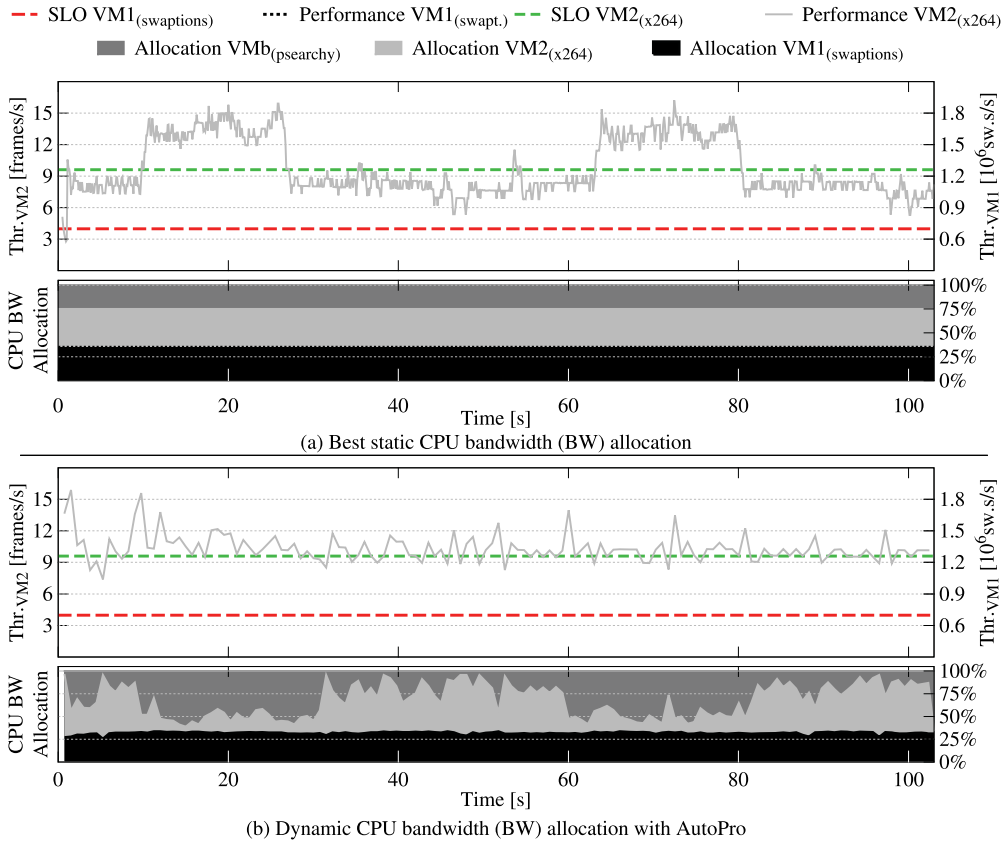

Fig. 11. Performance (throughput) and resource (CPU bandwidth) allocation traces for two SLO-bound VMs (VM1 and VM2), running *swaptions* and *x264*, respectively, colocated with a batch VM (VMb), running *psearchy*. The values of MAPE on the SLOs for the two VMs are, respectively, $\epsilon_1 = 0.3\%$ and $\epsilon_2 = 27.4\%$ with static allocation and $\epsilon_1 = 3.2\%$ and $\epsilon_2 = 7.4\%$ with *AutoPro*.

paradigm: with *AutoPro*, users specify an application-specific performance metric to state an SLO (e.g., requests/s for web server) and leave the task of determining resource needs to the virtualization infrastructure. In this section, we survey, to the best of our knowledge, related works; where possible, we provide qualitative comparisons with *AutoPro* and highlight both strengths and limitations of our work.

Padala et al. [2009] proposed *AutoControl*, an auto-scaling solution combining a model estimator and a set of application and node controllers. Similarly to *AutoPro*'s *Model Updater* (see Section 3.2.1), *AutoControl*'s model estimator captures the relationship between application performance and resource allocations (i.e., the resource–performance model). *AutoControl* employs a set of multi-input multioutput (MIMO) application controllers demanding node controllers to allocate CPU and I/O bandwidth, whereas *AutoPro* exploits single-input single-output (SISO) control targeting the bottleneck resource.

Shen et al. [2011] proposed *CloudScale*, an auto-scaling solution to allocate CPU bandwidth and memory, migrate VMs, and save energy through dynamic voltage and frequency scaling (DVFS). *CloudScale* leverages a system called *PRESS* [Gong et al. 2010] to handle resource demand prediction and prediction error. *PRESS* and *AutoPro*'s *Model Updater* carry on similar duties, as they both predict resource demands.

CloudScale couples *PRESS* with ad hoc heuristics to decide resource allocations; instead, *AutoPro* uses *VM Controllers* derived according to control theory from a resource–performance model, coordinated by the resource broker.

We acknowledge that our control strategy is currently limited in the number of resources it handles (the same holds for *CloudScale*, which handles resources disjointly). However, we believe that *AutoPro* can be extended to handle multiple resources by adding an additional layer to identify the bottleneck resource based on utilization, without incurring the overhead of solving a minimization problem of exponential complexity (as happens with *AutoControl*).

We argue that our evaluation is more compelling than the one proposed by Padala et al. [2009] and Shen et al. [2011] for the following reasons. First, we always report the error distributions of statistically relevant experimental results, allowing a clear analysis of these data. *AutoControl* and *CloudScale* only report single runs or average results, leaving much to be guessed. Second, we validate our resource–performance model with random resource allocations both in isolation and consolidation, while *AutoControl* only uses model-driven resource allocations in isolation and the heuristics used in *CloudScale* do not allow a similar validation. Finally, we collect experimental results with a much higher number of multi-VM workloads and SLOs and show that *AutoPro* achieves the additional goal of maximizing resource utilization. Furthermore, our control strategy works at a much higher frequency (i.e., control period <1 second instead of >10 seconds), giving *AutoPro* the ability of catching short-term trends that go unobserved with *AutoControl* and *CloudScale*. Short-term trends may become more and more important in the future, as predicted by other recent research [Ousterhout et al. 2013].

Nathuji et al. [2010] proposed *Q-Clouds*, a cloud infrastructure embodying a novel billing solution for public IaaS clouds and a runtime system that aims at mitigating the negative effects of contention on shared hardware resources; the design of *Q-Clouds* resembles that of *AutoControl*. To exploit unused resources, *Q-Clouds* allows the users to state additional SLOs (dubbed *Q-States*) and proportionally redistributes leftover capacity. Instead, *AutoPro* supports the execution of batch VMs on a best-effort basis and it is robust against the performance degradation these colocated VMs may introduce. *AutoPro*'s resource broker could support an equivalent of *Q-states* by redistributing (part of) the leftover capacity to SLO-bound VMs based on additional SLO levels.

Kocoloski et al. [2012] proposed a dual stack virtualization infrastructure to support the consolidation of general-purpose applications and high-performance computing (HPC) applications. They use two hypervisors: KVM for general-purpose applications and *Palacios* for HPC applications. Their approach to achieve isolation is trivial, consisting of partitioning a dual-socket host node between the two hypervisors: KVM manages a nonuniform memory access (NUMA) domain (i.e., a multicore with its cache hierarchy, prefetchers, memory controller, and memory banks), while *Palacios* manages another NUMA domain. Instead, *AutoPro* is much finer grained, as it considers multi-VM workloads sharing a single multicore (i.e., NUMA domain).

ControlWare [Zhang et al. 2002], *METE* [Sharifi et al. 2011], and *PTRADE* [Hoffmann et al. 2013] served as inspiration for subsystems of *AutoPro*. These systems tackle environments that differ from virtualization infrastructures, making a full-system comparison with *AutoPro* unfeasible; instead, we outline quantitative comparisons between subsystems. The error distributions for our resource–performance model (see Figure 5) display *maximum* values of MAPE in line with the *average* MAPEs of *METE*. The error distributions for *AutoPro* (see Figure 7) across a variety of applications with SLO 50% of the reference throughput are in line with the average error of *PTRADE* on the same applications; in addition, we evaluate a wider range of SLOs.

6. DISCUSSION AND FUTURE WORK

This section expands the discussion to highlight the strengths and limitations of *AutoPro* and point at directions for future work.

6.1. Multiple Resources and Shifting Bottlenecks

While the current implementation of *AutoPro* provisions CPU bandwidth and cores and manages compute-intensive multithreaded applications, the architecture we described is general enough to support applications with bottleneck resources other than the CPU; the lone requirement is to use an appropriate resource–performance model.

Our current implementation of *AutoPro* is not designed to manage deployments where different resources alternate as the bottleneck (e.g., a multitier web application split into a front-end web/application server and a back-end database management system can alternate compute- and I/O-bound execution phases [Padala et al. 2009]). In this case, *AutoPro* could leverage different resource–performance models, *VM Controllers,* and *Resource Brokers* for the different resources. To support this scheme, we plan to borrow ideas from *Dominant Resource Fairness* [Ghodsi et al. 2011] and extend the current implementation of the *Resource Broker*.

Some resources are tightly coupled on current multicores (e.g., CPU bandwidth and shared LLC usage) and do not lend themselves well to separate allocation. *AutoPro* already deals with this situation by soft partitioning the unmanaged resources through adjustments of the bottleneck resource allocation [Fedorova et al. 2007; Sironi et al. 2012]. Though not very efficient, this is a feasible solution to deal with those resources that cannot be easily partitioned on commodity architectures (e.g., shared caches, memory bandwidth).

The obvious (and, in general, more efficient) alternative to soft partitioning is to actually partition and explicitly allocate other resources. While we already discussed possible ways to extend *AutoPro* toward managing multiple resources, we need practical partitioning mechanisms as the enabling technology. For instance, while commodity systems do not expose direct methods to partition the LLC, we are working on shared LLC partitioning by means of page coloring [Lin et al. 2008]. A preliminary evaluation of *Rainbow* [Scolari et al. 2013], our *colored* page allocator extension for KVM, looks promising and could improve the efficiency of *AutoPro* by eliminating contention on the LLC. Other researchers have recently demonstrated that, with some limitations, it is possible to partition and allocate memory bandwidth [Yun et al. 2013]. We are planning to build on this research to extend *AutoPro*.

6.2. Different Performance Metrics

In this article, we present and evaluate a resource–performance model for throughput-driven applications; this model is applicable to any application that allows expressing a performance SLO as a throughput measurement; the seven applications we use in our evaluation show a varied sample of such applications. While throughput is an important metric, it does not fully capture the performance concerns of some applications; particularly, request-serving applications may be bound to an SLO on tail latency [Dean and Barroso 2013].

While the overall infrastructure of *AutoPro* is applicable to different performance metrics, supporting SLOs that are not expressed as a throughput target requires a different resource–performance model and, possibly, a different structure for the VM controllers. In particular, since nonlinearity is stronger in the resource-allocation-to-tail-latency relationship than in the resource-allocation-to-throughput relationship, our linear model would probably not be accurate if used to deal with latency-driven applications. Moreover, the monitoring system needs to be adapted to report the latency

at a given percentile instead of the average throughput over a time window. We are currently investigating these issues, which we believe are an important direction for future work.

An additional challenge that needs to be addressed to extend *AutoPro* to deal with latency-driven application lies at the resource-provisioning level. In our evaluation, we use a 750ms control period and do not go faster due to the increasing allocation inaccuracy of the *cpu* subsystem of *cgroups* with decreasing control periods. While this control period is already much finer grained than previous works (see Section 5) and it is fast enough to respond to execution phases of our throughput-driven applications, it would probably be too slow to provide the desired quality of service to applications bound to SLOs of tens of milliseconds (or less) on tail latency [Leverich and Kozyrakis 2014].

6.3. Scalability to Manycores and Large-Scale Installations

Most of the PARSEC applications show near-optimal scalability up to the core count of our reference host node [Sasaki et al. 2012]. Our plan to deal with the scalability issues that applications might have with higher core counts (i.e., on large symmetric multiprocessing systems or manycores) is to provision CPU bandwidth and cores at the same time. *AutoPro* could partition the available cores into islands (i.e., groups of cores), cluster applications with similar scalability (i.e., speedup over number of cores), assign each cluster to an island of the appropriate size, and allocate CPU bandwidth within each island.

We believe that the architecture of *AutoPro* can be extended to manage large-scale installations and distributed applications, such as those leveraging the MapReduce computation model [Dean and Ghemawat 2004]. Our efficient user-space implementation of the Application Heartbeats API [Hoffmann et al. 2010; Sironi et al. 2012] already supports distributed applications: tasks belonging to the same distributed applications share a multicast address and keep application-specific throughput measurements up to date through asynchronous messages.

6.4. Billing Scheme

A possible concern about *AutoPro* is its robustness to malicious users that make their VMs report falsely low performance with the goal of fooling the resource–performance model and getting additional resources. While this concern is legitimate, *AutoPro* is immune to such attack if the billing scheme used by the provider is based on resource consumption [Agmon Ben-Yehuda et al. 2012]. Using such a billing scheme with *AutoPro* raises two issues, both of which are practically solvable:

(1) It is hard for users to make a good cost estimate, as they do not explicitly rent resources.
(2) Responding to performance degradation through soft partitioning (see Section 6.1) increases resource usage, possibly leading to unfair billing.

To solve the first issue, the provider could ask users to state an upper bound to the resources they are willing to pay for and define an agreement on how to manage violations of this bound. To deal with the second issue, the provider could employ techniques to limit performance degradation [Mars et al. 2011; Govindan et al. 2011] and, when this is not enough, to estimate the extent of the performance degradation [Breslow et al. 2013; Dwyer et al. 2012] and scale the bill accordingly.

7. CONCLUDING REMARKS

Our experimental campaign (Section 4) shows with statistically relevant data that *AutoPro* achieves its two major goals:

(1) extending IaaS clouds with automated fine-grained resource allocation, asking users to just state the performance level they require for their VMs and
(2) enabling the cloud provider to safely share hardware resources among the VMs, allowing to to optimize data center utilization (thus reducing the total cost of ownership) by maximizing node-level utilization.

While the current implementation of *AutoPro* leaves few open issues that we are still investigating, the discussion of Section 6 outlines concrete solutions to fill these gaps.

We believe that *AutoPro*, by allowing one to safely share hardware resources and by operating at very fine scale (i.e., with subsecond control period and fine allocation granularity), is well suited for supporting better management of modern dynamic deployments; we hope that this proof of concept will help to improve the efficiency of our data centers.

ACKNOWLEDGMENTS

The authors would like to thank Gianluca C. Durelli and Martina Maggio for their early contributions to *AutoPro*; the anonymous reviewers of PACT '13, SoCC '13, VEE '14, and ACM Trans. Archit. Code Optim. for their valuable feedback on the various revisions of this article; and Harshad Kasture for his timely help in polishing the final revision.

Davide B. Bartolini was supported by a Ph.D. scholarship funded by the Italian government and the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) at the Politecnico di Milano. Filippo Sironi was partially supported by the European Union's Seventh Framework Programme (FP7/2007–2013) under grant agreement n. 287804.

The hexa-core host node used in our evaluation was donated by the Carbon Research Group, lead by Prof. Anant Agarwal at the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology (MIT), and was brought to Milano by Charles Gruenwald III via a winding path that passed through Taiwan.

REFERENCES

Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. 2012. The resource-as-a-service (RaaS) cloud. In *Proceedings of the 4th Workshop on Hot Topics in Cloud Computing (HotCloud'12)*. USENIX Association, Berkeley, CA.

Karl Johan Åström and Björn Wittenmark. 2008. *Adaptive Control*. Dover Publications.

Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*. USENIX Association, Berkeley, CA, 45–58.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 164–177. DOI:http://dx.doi.org/10.1145/945445.945462

Davide B. Bartolini, Riccardo Cattaneo, Gianluca C. Durelli, Martina Maggio, Marco D. Santambrogio, and Filippo Sironi. 2013a. The autonomic operating system research project: Achievements and future directions. In *Proceedings of the 50th Design Automation Conference (DAC'13)*. ACM, New York, NY, 77:1–77:10. DOI:http://dx.doi.org/10.1145/2463209.2488828

Davide B. Bartolini, Filippo Sironi, Martina Maggio, Gianluca C. Durelli, Donatella Sciuto, and Marco D. Santambrogio. 2013b. Towards a performance-as-a-service cloud. In *Proceedings of the 4th Symposium on Cloud Computing (SoCC'13)*. ACM, New York, NY, 26:1–26:2. DOI:http://dx.doi.org/10.1145/2523616.2525933

Christian Bienia. 2011. *Benchmarking Modern Multiprocessors*. Ph.D. Dissertation. Princeton University.

Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An analysis of linux scalability to many cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, 1–16.

Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. 2013. Enabling fair pricing on HPC systems with node sharing. In *Proceedings of the International Conference*

on High Performance Computing, Networking, Storage and Analysis (SC'13). ACM, New York, NY, 37:1–37:12. DOI: http://dx.doi.org/10.1145/2503210.2503256

Lydia Y. Chen, Danilo Ansaloni, Evgenia Smirni, Akira Yokokawa, and Walter Binder. 2012. Achieving application-centric performance targets via consolidation on multicores: Myth or reality?. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'12)*. ACM, New York, NY, 37–48. DOI: http://dx.doi.org/10.1145/2287076.2287083

Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*. ACM, New York, NY, 308–319. DOI: http://dx.doi.org/10.1145/2485922.2485949

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. DOI: http://dx.doi.org/10.1145/2408776.2408794

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. USENIX Association, Berkeley, CA, 137–149.

Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. 2012. A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*. IEEE Computer Society Press, Los Alamitos, CA, 83:1–83:11. DOI: http://dx.doi.org/10.1109/SC.2012.11

Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. 2012. More for your money: Exploiting performance heterogeneity in public clouds. In *Proceedings of the 3rd Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY, 20:1–20:14. DOI: http://dx.doi.org/10.1145/2391229.2391249

Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. 2007. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*. IEEE Computer Society, Washington, DC, 25–38. DOI: http://dx.doi.org/10.1109/PACT.2007.40

Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, 323–336.

Daniel Gmach, Jerry Rolia, and Ludmila Cherkasova. 2012. Selling t-shirts and time shares in the cloud. In *Proceedings of the 12th International Symposium on Cluster, Cloud and Grid Computing (CCGRID'12)*. IEEE Computer Society, Washington, DC, 539–546. DOI: http://dx.doi.org/10.1109/CCGrid.2012.68

Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. 2010. PRESS: PRedictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Services Management (CNSM'10)*. International Federation for Information Processing, Laxenburg, Austria, 9–16. DOI: http://dx.doi.org/10.1109/CNSM.2010.5691343

Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd Symposium on Cloud Computing (SoCC'11)*. ACM, New York, NY, 22:1–22:14. DOI: http://dx.doi.org/10.1145/2038916.2038938

Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley & Sons.

Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. 2010. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC'10)*. ACM, New York, NY, 79–88. DOI: http://dx.doi.org/10.1145/1809049.1809065

Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. 2013. A generalized software framework for accurate and efficient management of performance goals. In *Proceedings of the 11th International Conference on Embedded Software (EMSOFT'13)*. IEEE Press, Piscataway, NJ, 19:1–19:10. DOI: http://dx.doi.org/10.1109/EMSOFT.2013.6658597

Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger. 2013. Efficient live migration of virtual machines using shared storage. In *Proceedings of the 9th International Conference on Virtual Execution Environments (VEE'13)*. ACM, New York, NY, 41–50. DOI: http://dx.doi.org/10.1145/2451512.2451524

Harshad Kasture and Daniel Sanchez. 2014. Ubik: Efficient cache sharing with strict Qos for latency-critical workloads. In *Proceedings of the 18th International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS'14). ACM, New York, NY, 729–742. DOI:http://dx.doi.org/10.1145/2541940.2541944

Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: The Linux virtual machine monitor. In *Proceedings of the Linux Symposium*. 225–230.

Brian Kocoloski, Jiannan Ouyang, and John Lange. 2012. A case for dual stack virtualization: Consolidating HPC and commodity applications in the cloud. In *Proceedings of the 3rd Symposium on Cloud Computing (SoCC'12)*. ACM, New York, NY, 23:1–23:7. DOI:http://dx.doi.org/10.1145/2391229.2391252

Jacob Leverich and Christos Kozyrakis. 2014. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys'14)*. ACM, New York, NY.

Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA'08)*. IEEE Computer Society, Washington, DC, 367–378. DOI:http://dx.doi.org/10.1109/HPCA.2008.4658653

Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th International Symposium on Microarchitecture (MICRO'11)*. ACM, New York, NY, 248–259. DOI:http://dx.doi.org/10.1145/2155620.2155650

David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2011. The PowerNap server architecture. *ACM Trans. Comput. Syst.* 29, 1 (Feb. 2011), 3:1–3:24. DOI:http://dx.doi.org/10.1145/1925109.1925112

Paul Menage. 2013. CGROUPS. Available at http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt.

Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-Clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 237–250. DOI:http://dx.doi.org/10.1145/1755913.1755938

Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. 2009. Analytical modeling of pipeline parallelism. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. IEEE Computer Society, Washington, DC, 281–290. DOI:http://dx.doi.org/10.1109/PACT.2009.28

Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, low latency scheduling. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP'13)*. ACM, New York, NY, 69–84. DOI:http://dx.doi.org/10.1145/2517349.2522716

Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated control of multiple virtualized resources. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys'09)*. ACM, New York, NY, 13–26. DOI:http://dx.doi.org/10.1145/1519065.1519068

Conrad Sanderson. 2010. *Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments*. Technical Report. NICTA.

Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. 2012. Scalability-Based manycore partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 107–116. DOI:http://dx.doi.org/10.1145/2370816.2370833

Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys'13)*. ACM, New York, NY, 351–364. DOI:http://dx.doi.org/10.1145/2465351.2465386

Alberto Scolari, Filippo Sironi, Davide B. Bartolini, Donatella Sciuto, and Marco D. Santambrogio. 2013. Coloring the cloud for predictable performance. In *Proceedings of the 4th Symposium on Cloud Computing (SoCC'13)*. ACM, New York, NY, 47:1–47:2. DOI:http://dx.doi.org/10.1145/2523616.2525955

Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. 2011. METE: Meeting end-to-end QoS in multicores through system-wide resource management. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'11)*. ACM, New York, NY, 13–24. DOI:http://dx.doi.org/10.1145/1993744.1993747

Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd Symposium on Cloud Computing (SoCC'11)*. ACM, New York, NY, 5:1–5:14. DOI:http://dx.doi.org/10.1145/2038916.2038921

Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. 2012. Metronome: Operating system level performance management via self-adaptive computing. In *Proceedings of the 49th Design Automation Conference (DAC'12)*. ACM, New York, NY, 856–865. DOI:http://dx.doi.org/10.1145/2228360.2228514

Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. 2014. A performance-aware quality of service-driven scheduler for multicore processors. *SIGBED Rev.* 11, 1 (Feb. 2014), 50–55. DOI:http://dx.doi.org/10.1145/2597457.2597464

Standard Performance Evaluation Corporation. 2013. SPECvirt_sc2013. Available at http://www.spec.org/virt_sc2013/. (2013).

Paul Turner, Bharata B. Rao, and Nikhil Rao. 2010. CPU bandwidth control for CFS. In *Proceedings of the Linux Symposium*. 245–254.

Xen Project. 2013. Credit Scheduler – Xen. Available at http://wiki.xen.org/wiki/CreditScheduler. (2013).

Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. 2013. MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. IEEE Computer Society, Washington, DC, 55–64. DOI:http://dx.doi.org/10.1109/RTAS.2013.6531079

Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. 2002. ControlWare: A middleware architecture for feedback control of software performance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*. IEEE Computer Society, Washington, DC, 301–310. DOI:http://dx.doi.org/10.1109/ICDCS.2002.1022267