Scale & Cap: Scaling-Aware Resource Management for Consolidated Multi-threaded Applications

CAN HANKENDI and AYSE KIVILCIM COSKUN, Boston University

As the number of cores per server node increases, designing multi-threaded applications has become essential to efficiently utilize the available hardware parallelism. Many application domains have started to adopt multi-threaded programming; thus, efficient management of multi-threaded applications has become a significant research problem. Efficient execution of multi-threaded workloads on cloud environments, where applications are often consolidated by means of virtualization, relies on understanding the multi-threaded specific characteristics of the applications. Furthermore, energy cost and power delivery limitations require data center server nodes to work under power caps, which bring additional challenges to runtime management of consolidated multi-threaded applications. This article proposes a dynamic resource allocation technique for consolidated multi-threaded applications for power-constrained environments. Our technique takes into account application characteristics specific to multi-threaded applications, such as power and performance scaling, to make resource distribution decisions at runtime to improve the overall performance, while accurately tracking dynamic power caps. We implement and evaluate our technique on state-of-the-art servers and show that the proposed technique improves the application performance by up to 21% under power caps compared to a default resource manager.

CCS Concepts: • **Hardware** → **Platform power issues**; • **Software and its engineering** → *Designing software;*

Additional Key Words and Phrases: Multi-threaded, multi-core, power, energy efficiency, virtual machines

ACM Reference Format:
Can Hankendi and Ayse Kivilcim Coskun. 2017. Scale & cap: Scaling-aware resource management for consolidated multi-threaded applications. ACM Trans. Des. Autom. Electron. Syst. 22, 2, Article 30 (January 2017), 22 pages.
DOI: http://dx.doi.org/10.1145/2994145

1. INTRODUCTION

As the demand on the cloud continuously increases, the efficient use of power and compute resources has become increasingly important. It is predicted that 70% of all workloads will be executed on cloud resources by 2015 [Cisco 2013]. In order to meet the increasing user demand, the number of servers installed on cloud resources has been tripled in the last decade [Borovick 2011]. Although increasing the number of servers can boost the compute capacity, power delivery and financial constraints limit the maximum achievable performance. In order to comply with power and cost constraints, power-capping techniques have become an essential feature for any data center [Cochran et al. 2011]. Independent system operators (ISOs) have started offering

Authors' addresses: C. Hankendi and A. K. Coskun, 8 St. Mary's St. Boston, MA, 02215; emails: {hankendi, acoskun}@bu.edu.

DOI: http://dx.doi.org/10.1145/2994145

significant cost savings for data centers that can dynamically comply with changing power constraints [Chen et al. 2013], further motivating dynamic power capping.

In addition to the limitations due to power delivery, the management of increasing the number of users becomes significantly complex for large-scale computing clusters. Therefore, virtualization has become another important standard for large-scale computing systems, as it provides flexible and efficient management of cloud resources through seamless consolidation. Consolidating multiple virtual machines (VMs) allows increasing overall utilization of the cloud resources, which significantly reduces the number of active server nodes that is required to meet the quality-of-service (QoS) requirements [Beloglazov and Buyya 2010]. However, consolidation brings additional challenges to cloud management. Consolidating multiple VMs on a single server may lead to increased resource contention, which, in turn, can hurt the performance. In order to reduce the resource contention due to consolidation, state-of-the-art VM placement techniques use metrics that measure the memory activity to find the best matching VMs that minimally interfere with each other's execution [Dhiman and Rosing 2007].

Although placement techniques are critically important to minimize the resource contention due to consolidation, they lack the ability to manage the available resources on a given server to optimize the performance or comply with the power constraints. While there is a significant amount of work in literature to address the consolidation and resource distribution problem under power constraints for single-threaded workloads [Nathuji and Schwan 2008; Hankendi et al. 2013; Reda et al. 2012], the emergence of multi-threaded workloads on cloud resources and related challenges require these techniques to be revisited. Ideally, multi-threaded applications are designed to exhibit linear performance improvement and power consumption with increasing number of threads when a sufficient amount of resources are available. However, communication and synchronization overheads and microarchitectural bottlenecks might lead to sublinear performance improvement, which needs to be considered while making resource allocation decisions. For example, allocating more resources to an application with higher power efficiency may improve overall throughput in a consolidated server, assuming equivalent priorities and QoS constraints for the applications sharing the server.

In this article, we propose a dynamic resource allocation technique for consolidated environments that run multi-threaded applications while operating under power caps. Our work differentiates from prior research allocation methods [Hankendi et al. 2013; Dhiman and Rosing 2007; Isci et al. 2010] by (1) jointly considering the application power and performance scalability information to make decisions, (2) introducing a formal linear programming–based solution that is able to manage various numbers of VMs under a wide range of power and performance constraints, and (3) demonstrating the relative benefits of VM placement and server-level resource allocation algorithms under power-constrained operation. We implement our runtime resource allocation technique on state-of-the-art servers with multi-core processors. Our specific contributions are as follows:

—We analyze existing dynamic resource allocation techniques [Isci et al. 2010; Dhiman and Rosing 2007; Hankendi et al. 2013] and show their shortcomings for multi-threaded applications.
—We propose a dynamic resource allocation technique that incorporates both power and performance scaling characteristics of multi-threaded applications to improve the overall performance under power caps, while providing QoS guarantees. We implement a linear programming (LP)–based solution on a separate management node to make resource allocation decisions at runtime on multiple state-of-the-art multi-core servers.

—We evaluate our technique together with a set of modern placement algorithms and identify the performance benefits due to placement and resource allocation decisions. Our results show that our technique improves the performance by up to 24% with respect to the default resource manager of the hypervisor, while meeting the power constraints at 98% of the time within a ±2W error range.

Section 2 presents two case studies that motivate the idea of jointly using power and performance scalability information to improve performance under power caps. Section 3 introduces **Scale & Cap**, which is a linear programming–based formal solution for resource allocation problem under power and performance constraints. Section 4 provides the details of our experimental setup. Section 5 presents the benefits of **Scale & Cap** when integrated with state-of-the-art VM placement techniques. Section 6 discusses the state-of-the-art solutions for VM placement, resource allocation, and power capping, and Section 7 concludes the article.

2. IMPACT OF APPLICATION POWER AND PERFORMANCE SCALABILITY ON CONSOLIDATION EFFICIENCY

The total amount of available compute resources on a server varies over time due to the power cap given to the server, thermal emergencies, user demands, and application types. Traditionally, CPU utilization has been used as the metric to determine the resource distribution ratio across single-threaded applications [Nathuji et al. 2009]. By using the CPU utilization metric to distribute the resources proportionally, these techniques aim to minimize the performance degradation while maximizing the server utilization. However, with the emergence of multi-threaded applications, using a single dimensional metric, such as CPU utilization, becomes inefficient to capture the multi-threaded specific characteristics. In this section, we first present two motivational examples to show the need for a novel approach to the resource allocation problem for multi-threaded applications. We then introduce our dynamic resource allocation technique that jointly uses the power and performance characteristics to capture the multi-threaded characteristics.

2.1. Is CPU Utilization Enough?

CPU utilization metric measures the percentage of busy cycles of a specific period of time. Traditional resource allocation techniques distribute the available compute resources proportional to the CPU utilization levels of the consolidated applications [Xen 2011; VMware 2013]. Although, for single-threaded applications, CPU utilization can capture all CPU requirements of an application; for multi-threaded applications, we need to incorporate an additional dimension, which is the performance scalability. Performance scalability can be defined as the characteristics of a multi-threaded application that reflect how the performance of the application is increasing (i.e., scaling) with increased amount of resources. Ideally, all multi-threaded applications are designed to scale perfectly with an increasing amount of resources (i.e., $2x$ performance improvement for $2x$ increase in allocated resources). However, due to various multi-threaded application bottlenecks, such as communication, synchronization, serial code, and the like, most of the multi-threaded applications do not scale linearly with an increasing amount of resources. Therefore, each multi-threaded application has a specific performance/resource curve that reflects its performance scalability.

On the ESXi hypervisor, the total available computational capacity of a server node is represented in MHz, where the total amount of CPU resources, R, is equal to the number of physical CPUs multiplied by the maximum core frequency. CPU resource usage of a VM can be constrained by adjusting the *CPU resource limits* on the ESXi hypervisor. Figure 1 shows the QoS scaling of four applications from PARSEC and

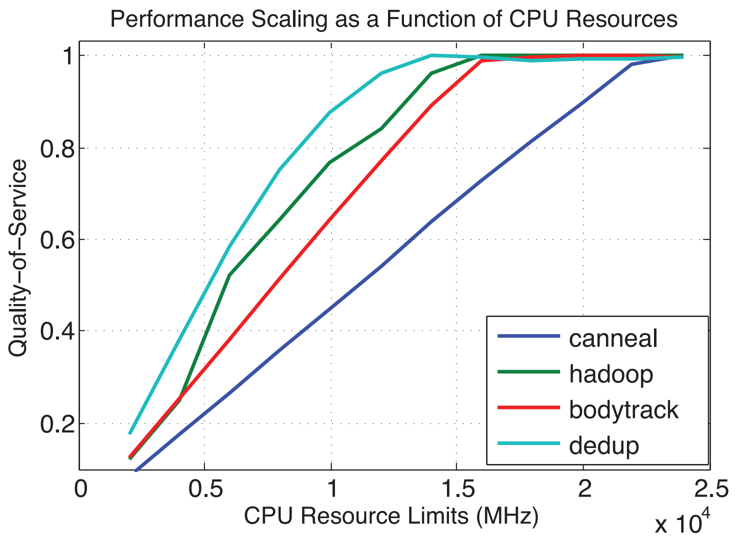

Fig. 1. Performance scaling of some of the PARSEC benchmarks and `hadoop` from Cloudsuit as a function of CPU resource limits.

Cloudsuite as a function of the CPU resources (in MHz). As Figure 1 shows, `bodytrack` cannot utilize all the available hardware resources; therefore, its QoS does not improve beyond a certain amount of CPU resources (i.e., 15,970MHz). In addition, reducing the CPU resources has a larger performance impact on the poorly scaling VMs (e.g., canneal, bodytrack) at lower CPU resource limits. Therefore, considering the performance scalability while making resource allocation decisions is expected to have a substantial impact on the overall performance of the system.

Similar to the performance scalability characteristics, each application has a distinct peak power consumption and a power weight, w_i, which represents the power consumed at one unit of computing capacity (i.e., MHz in the virtual environment), while running application i. In Figure 2, we show the peak power consumption and the power weights of 14 applications and the average value for all applications. In order to obtain the peak power values, we run the applications alone with maximum amount of thread counts that are equal to the total core count for each server (i.e., 12 threads for the AMD-based and 8 threads for the Intel-based server) with the default resource manager. As the figure shows, both peak power and power weight values show significant variation. Similar to our argument for performance scalability, resource distribution without considering the power weights of the individual applications can lead to inefficient resource distribution by not being able to favor more power-efficient applications. In order to better illustrate our observations, we present two case studies that demonstrate the benefits of considering performance scalability and power weights while making resource distribution decisions.

2.2. Case Study #1: Applications with Distinct Performance and Power Scaling

In this example, we compare the benefits of various resource distribution approaches on two applications that do not exhibit only distinct resource requirements (i.e., performance scaling), but also distinct power characteristics. Such an application-pair from PARSEC suite is `canneal` and `facesim`. The performance of `canneal` can scale almost up to 12-threads, while `facesim`'s performance increase saturates beyond 8-threads. On the contrary, `facesim` is a more power-hungry application when compared to `canneal` due

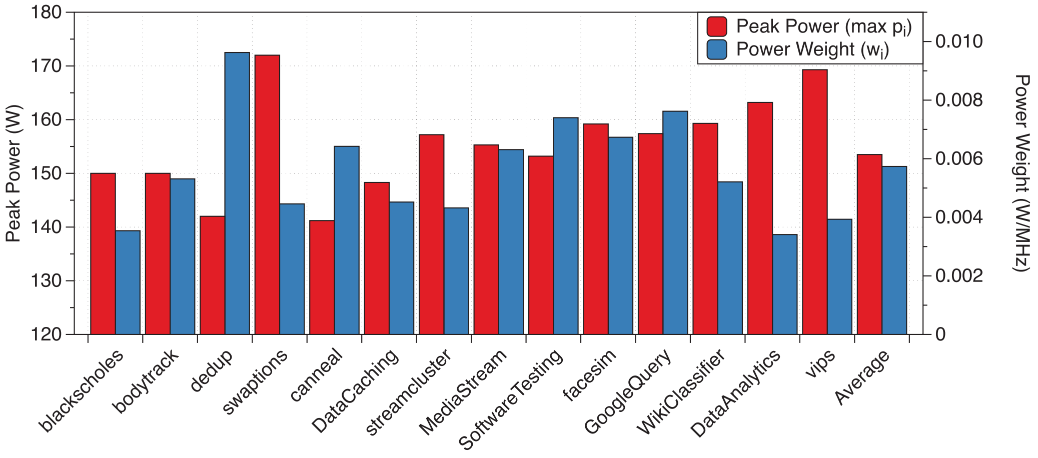

Fig. 2. Peak power (left axis, red bars) and power weight (right axis, blue bars) values for four PARSEC benchmarks and the PARSEC average (right-most bars) measured on AMD Opteron 6172.

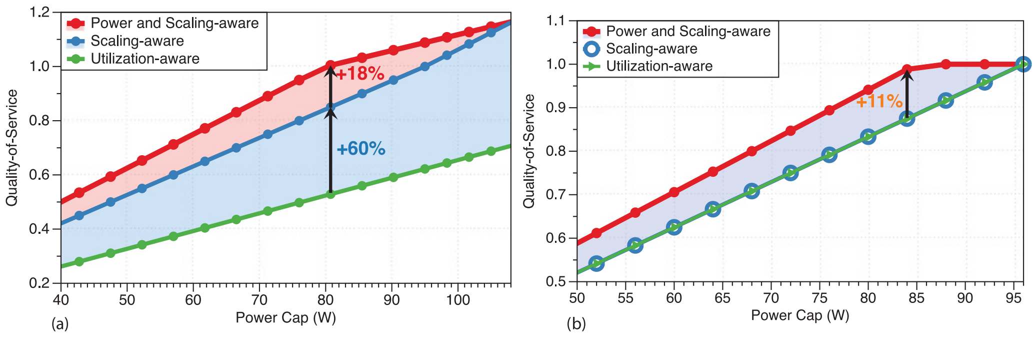

Fig. 3. Total QoS comparison for consolidating (a) `canneal-facesim` and (b) `blackscholes-swaptions` on AMD Opteron 6172 with various power caps for the utilization-based approach (baseline), naive approach (only scaling-aware), and power-aware (scaling and power-aware) approach. Power-awareness brings up to 18% and 11% QoS improvements over the scaling-only approach.

to its higher Instruction-per-cycle (IPC) [Hankendi and Coskun 2012]. These contradicting properties play a significant role while making resource distribution decisions. In order to illustrate this, we compare three algorithms: (1) utilization-aware [Nathuji and Schwan 2008], (2) only performance scalability-aware [Hankendi et al. 2013], and (3) power and performance scalability-aware in Figure 3(a). The utilization-aware approach proportionally distributes the total available amount of resources based on the CPU utilization of each VM by calculating a weight value (w_i) as shown in Algorithm 1. The scaling-aware resource distribution first estimates the CPU requirements of each VM and then prioritizes the ones with smaller requirements to maximize the overall QoS of the server, as shown in Algorithm 2. On the other hand, the power and performance scaling-aware approach maximizes the QoS for given power weights, CPU demands, power constraints, and the total amount of available resources using a linear programming–based solver, which is explained in detail in Section 3. As the power-aware approach uses power weights as an additional metric, it can further improve the overall QoS by guiding its decision based on both resource and power constraints.

ALGORITHM 1: Utilization-aware Resource Distribution

```
Inputs:
$U[n]$ // Utilization Array
$R_k$ // Total amount of available resources
Output: $r_i$
$W = sum(U[n])$;
for i = 1 to n;
$w_i = U[i]/W$
$r_i = R_k * w_i$
end
```

ALGORITHM 2: Scalability-aware Resource Distribution

```
Inputs:
$C[n]$ // CPU Demand Array
$R_k$ // Total amount of available resources
Output: $r_i$
sort: C[n] for i = 1 to n;
if $R_k > 0$ then
    $r_i = C[i]$;
    $R_k = R_k - r_i$;
else
    return
end
end
```

2.3. Case Study #2: Applications with Similar Performance but Distinct Power Scaling

In order to show the benefits of adding power considerations to the resource distribution technique, we evaluate two applications from PARSEC benchmark suite that exhibit similar performance scalability characteristics, but distinct power requirements, such as `blackscholes` and `swaptions`. Although both of these applications can scale up to 12-threads, `swaptions` consumes significantly higher power than `blackscholes` (e.g., 151W vs. 172W). We use the same baselines as the previous case study.

In Figure 3(b), we show the total QoS of consolidating `blackscholes` and `swaptions` under various power caps for three approaches: power and scaling-aware, only scaling-aware, and utilization-based (baseline). As Figure 3(b) shows, utilization-aware and scaling-aware techniques perform exactly the same, as the performance scalability of these two applications can not be distinguished. Therefore, only scaling-aware resource distribution equally favors these two applications, as they have similar performance scaling capacity. On the other hand, the power and scaling-aware approach favors the one with lower power weight value. The power and scaling-aware approach improves the total QoS up to 11% in comparison to only scaling-aware and utilization-based approaches. As `blackscholes` and `swaptions` have very similar performance scaling behavior, scaling-awareness does not bring any benefits over the utilization-based approach. This also demonstrates that the 11% performance improvement is solely due to power-awareness.

3. SCALE & CAP: LINEAR PROGRAMMING–BASED DYNAMIC RESOURCE ALLOCATION

Based on our observations in the previous section, we conclude that the power efficiency and performance scaling characteristics of parallel applications play a significant role. In order to incorporate our findings into a formal solution, we formulate the problem as a linear programming problem. The main goal of the solution provided here is to maximize the total QoS of consolidated applications without violating the individual

QoS requirements of the applications under power constraints through resource allocation. We first formulate the problem of maximizing QoS for n number of VMs, then explain how to incorporate power efficiencies of individual applications into our linear programming solution.

We define the maximum QoS as the performance (i.e., runtime) of an application when running alone on the target system. This measurement gives us the upper-bound for the performance of each application, which we use as the maximum QoS of 1. Any performance loss is reflected as the percentage of the maximum QoS performance. The problem of maximizing QoS of a server when consolidating m applications with QoS values on a single physical server can be represented as follows.

$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^{m} q_i \\ &\text{subject to} \quad 0 \le q_i \le 1,\ i = 1, \dots, m. \end{aligned} \tag{1}$$

For an application i, the achievable maximum QoS (q_i) is a function of the resource demand, (d_i), and the total amount of resources allocated, (s_i). In order to achieve the maximum QoS, the amount of supplied resources (s_i) should not be lower than the resource demand of the application (d_i). Therefore, we can define the QoS of an application, i, as a function of resource demand (d_i) and the allocated/supplied resources (s_i). For the case where $d_i \le s_i$, QoS is 1, as the demand is met by the supply. Therefore, our focus is to solve the resource distribution problem, where resources are limited ($d_i \ge s_i$). For such cases, QoS of the application i, q_i is described as follows:

$$\begin{aligned} &q_i = s_i/d_i \\ &\text{subject to} \quad 0 \le s_i \le d_i \le R_k. \end{aligned} \tag{2}$$

R_k is the total resource capacity of a server, k. For consolidating $i = 1, \dots, m$ number of applications on a server, k, Equation (1) (i.e., the QoS maximization problem) becomes

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{m} s_i/d_i \\ &\text{subject to} \quad 0 \le s_i \le d_i \le R_k. \end{aligned} \tag{3}$$

In order to be able to guarantee certain performance requirements, we need to be able to enforce lower bounds for the QoS of each application. We can use *lower-bound* for minimum performance guarantees and *upper-bound* for maximum performance limitations, as data centers may provide incentives to users for bounding the maximum performance. Therefore, our problem becomes a constrained QoS maximization problem, which can be represented by putting lower and upper bound constraints on q_i (i.e., s_i/d_i) of the applications, such that

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{m} s_i/d_i \\ &\text{subject to} \quad 0 \le s_i \le d_i \le R_k \\ &\qquad\qquad\quad l_i \le q_i \le u_i. \end{aligned} \tag{4}$$

Based on the equations just given, we can convert the problem into a linear-programming problem as follows:

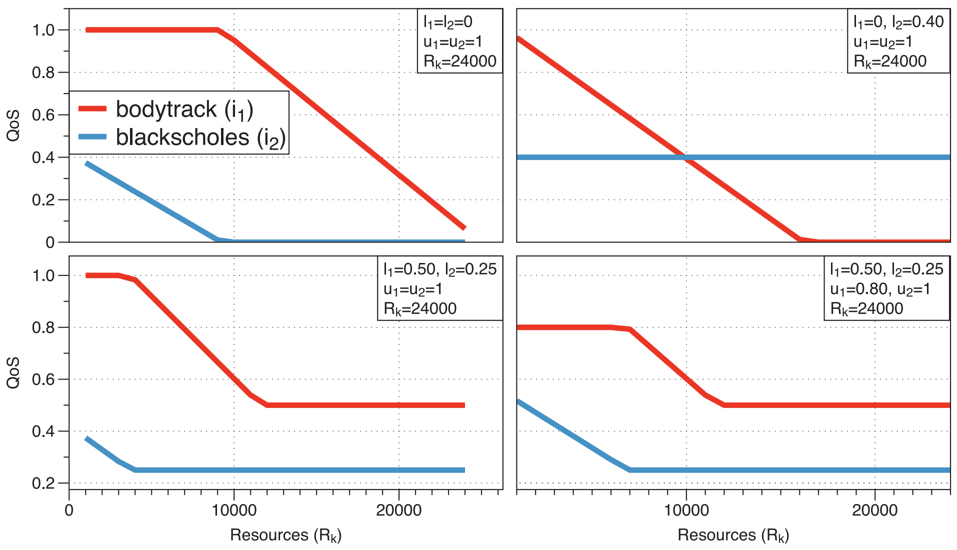

Fig. 4. LP-solution for resource distribution across two applications ($m = 2$) with various lower and upper bounds for a given amount of resources R_k.

Find **q** that maximizes

$$
\begin{aligned}
& f(q) = q_1 + q_2 + \dots q_m \\
\text{subject to} \quad & \sum_{i=1}^{m} d_i q_i \leq R_{max} \qquad (5)\\
& l_i \leq q_i \leq u_i .
\end{aligned}
$$

Solutions of this LP-problem provide us the **q** vector, which consists of a set of possible q_n values, $q_1, q_2, \dots q_m$, to assign for each application, i, to maximize the $f(q)$ (i.e., total QoS) under a total resource capacity constraint, R_{max}. For an unlimited amount of resources, R_{max}, the maximum value of $f(q)$ would be m, as $q_1, q_2, \dots, q_m = 1$. For an application, i, with a total resource demand d_i, the total amount of resources required to provide a QoS of q_i is, $s_i = d_i q_i$. From the LP solution, we can derive the necessary amount of resources, s_i, that maximizes the total QoS of the system (i.e., $f(x)$). Figure 4 shows the LP-solution for bodytrack and blackscholes benchmarks under various lower and upper QoS constraints.

3.1. Maximizing Server-QoS with Power Constraints

For the case where there are power constraints, the amount of resources needs to be reduced to R_k, where $R_k \leq R_{max}$. In order to determine the value of R_k, our runtime system utilizes the power feedback, P_t, and the system utilization measurement window, $R(t)$, which is the running average of the last four resource demand estimates, $R[t_{n-3}, \dots, t_n]$. For a given time, t, and power constraint at time t, $P_{cap}(t)$, it is possible to derive the $R_k(t)$ by using the linear correlation between $R(t)$ and $P(t)$. Based on our experimental results and reported results from prior work, we assume that power constraints on the system are linear and can be derived at runtime through power measurement feedback. As a naive approach that allows us to meet power constraints through modifying R_k, we simply compute $R_k(t)$ based on the $P(t)$ for a given power constraint, $P_{cap}(t)$ by

$$R_k(t) = P_{c,k}(t)/P_k(t-1) * R_k(t-1) \qquad (6)$$

Table I. Definitions of the Abbreviations Used in the LP Solution

Abbreviation	**Definition**
P_k	Power consumption of a server, k.
p_i	Power consumption estimation of an application, i.
R_k	Total amount of resources allocated to a server.
r_i	Total amount of resources allocated to an application, i.
W_k	Power weight of a consolidated application set, i.
w_i	Power weight of an individual application, i.
u_i	Performance upper bound for application, i.
l_i	Performance lower bound for application, i.
k	Server index
i	Workload index

By using the most recent power weight $W_k(t-1)$, it is possible to find an $R_k(t)$ that consumes $P_k(t) \leq P_{c,k}(t)$. After deriving the $R_k(t)$ value to be enforced on the server, the QoS maximization problem can be solved by solving the problem represented in Equation (5). However, this approach simply uses a lumped value, W_k, for modeling the power/performance relation of a set of applications, where each individual application has a distinct power weight value, w_i. Therefore, any change on r_i has a different impact on p_i, and therefore P_k, where, $P_k = \sum_{i=1}^{m} p_i$. Ignoring the power weight differences across applications leads to inefficient resource distribution as illustrated in Section 2.3. In order to derive the power weights of the applications, we use VM-level power estimations on the Intel-based server. As VM-level power estimations are not available for the AMD-based server, we use offline data for the AMD-based server. For the Intel-based server, power weights (w_i) values can be derived at runtime by $P_{VM_i}(t)/R_{VM_i}$. We incorporate the power weight (w_i) information into the LP solution as shown in Equation (7).

$$\begin{aligned} & f(q) = q_1 + q_2 + \ldots q_m \\ \text{subject to} \quad & \sum_{i=1}^{m} s_i \leq R_k \\ & \sum_{i=1}^{m} s_i * w_i \leq P_{cap} \\ & l_i \leq q_i \leq u_i. \end{aligned} \tag{7}$$

As the power weight value (w_i) represents the power consumed-per-MHz computation, we estimate the power consumption by summing up the $s_i * w_i$ multiplication in Equation (7). By enforcing an additional constraint in the form of power constraints, we target to use the available resources and power as efficiently as possible. We list all variables and their definitions in Table I.

3.2. Runtime Implementation of Scale & Cap

We implement the LP solution in MATLAB and compile it as an executable file. In order to simplify the implementation, we convert the QoS maximization problem $(max f(n))$ to a minimization problem $(min f(-n))$. We then use *linprog* MATLAB routine to solve the minimization problem to find the $\boldsymbol{q}$. The inputs to the MATLAB routine are the user constraints for upper and lower-bounds, respectively u_i and l_i, power constraints (P_{cap}), power measurements from the power meter (P_k), and VM-level metrics from the hypervisor to derive w_i and r_i values. We show the overall flow of the runtime implementation in Figure 5. We evaluate runtime resource allocation techniques together

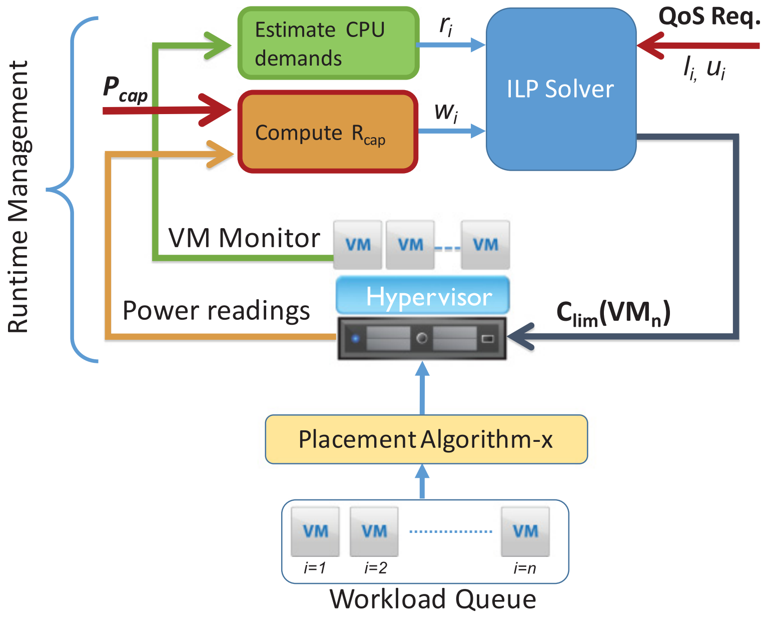

Fig. 5. Illustration of the runtime implementation. Runtime implementation consists of two stages: placement and resource allocation. At runtime, resource allocation decisions are determined through real-time power and performance monitoring.

with placement algorithms. We use offline workload characteristics to determine the VM placement decisions that are guided by the algorithms explained in Section 5.2. On the other hand, resource allocation decisions are determined at runtime using real-time power and VM-level performance monitoring.

We compiled the MATLAB module as a C library to call on a separate management node at runtime that communicates with the host through the VMware vCLI Perl API to change the allocated resources for each VM [VMware 2015]. The main control loop of the implementation runs every 2 seconds, which is the same as the `esxtop` sampling rate. Changing resource limits at this granularity imposes a negligible performance overhead on the applications. The LP solution routine takes between 0.2 to 0.4 seconds to return the updated r_i values. The API-based communication with the hypervisor completes its function within the range of 0.1 to 0.3 seconds. Therefore, within a 2 second window, the runtime implementation can finalize its decision and action. For faster control, it is also possible to implement the LP solution with alternative libraries and/or languages. However, note that for large-scale systems, the monitoring period is reported to be over 20 seconds [VMware 2013].

4. EXPERIMENTAL SETUP

As our main target is managing multi-core environments, our experimental setup includes two state-of-the-art servers: one with Intel Xeon E5 and the other with AMD Magny Cours (Opteron 6172) multi-core processors. Intel Xeon E5-2603 processors consist of 8 cores. Each core has 32KB of private L1 and 256KB of private L2 cache. All 8 cores share 10MB of L3 cache and 32GB memory. Magny Cours consists of two 6-core dies attached together on a single chip. Each 6-core die includes a 12MB shared L3 cache, and each core has a 512KB private L2 cache. All cores also share a 16GB off-chip memory. We virtualize both systems with the VMware ESXi 5.5 hypervisor.

We create VMs with multiple vCPUs (SMP VMs) such that each VM accommodates a multi-threaded application. Each VM runs Ubuntu 12.04 as the guest OS.

We run all 13 applications from the PARSEC multi-threaded benchmark suite [Bienia et al. 2008], 4 applications from Cloudsuite [Ferdman and et al. 2012], and 3 applications from the BigDataBench benchmark suite [Wang et al. 2014] in our experiments, as a representative set of multi-threaded workloads on the cloud resources. We track application-specific performance metrics for the PARSEC benchmarks by utilizing the Application Heartbeats framework [Hoffmann et al. 2011]. CloudSuite applications report application-specific performance without requiring a modification to the source code. We choose to evaluate application-specific performance metrics, because instruction count–based performance metrics do not always provide meaningful performance feedback to the user. For instance, for image processing applications (e.g., `bodytrack`) the QoS metric is *frames-per-second* (FPS), whereas the QoS metric for the option trading application (e.g., `blackscholes`) is the *number of options*. We report the relative QoS for each application, where we define the maximum QoS of an application (i.e., QoS=1) as the case where the application is running alone with the maximum amount of available CPU resources (e.g., maximum number of cores, no resource limits imposed by the hypervisor).

To measure VM-level CPU metrics, we utilize the `esxtop` utility that is available in the ESXi hypervisor. We sample the VM-level metrics every 2 seconds, which is the maximum sampling rate for `esxtop`. We measure the system power by using a *Wattsup PRO* power meter with a 1 second sampling rate. As the total system power determines the electricity cost of a server, we focus on the system power rather than the component power (i.e., processor, disk, etc.). The resource allocation decisions are handled by the hypervisor and the OS-level tools, as described in Section 3.

In all of our experiments, we only evaluate the parallel phases of the applications, as the parallel phase of multi-threaded applications dominates the application execution time in real-life clusters. We implement a consolidation management module that synchronizes the starting point of the parallel phases of the applications [Hankendi and Coskun 2012] and collects performance data only for the parallel phases, until one of the applications' parallel phases finishes.

5. EXPERIMENTAL RESULTS

In this section, we quantify the benefits of power and performance scaling-aware resource distribution across multiple VMs. We compare our technique with previously proposed resource distribution policies and present the performance improvements under various power caps. In addition to comparison among resource distribution techniques, we also compare the benefits from placement and resource distribution techniques to gain insights about the interaction as well as to provide quantitative comparisons between two resource management schemes.

5.1. Evaluating Resource Allocation Techniques

In order to quantify the benefits of our resource distribution technique, we choose three approaches that are already implemented or proposed in prior work, namely, the default ESXi manager, demand proportional distribution, and performance scaling proportional resource distribution [Hankendi et al. 2013]. We briefly explain each approach as follows:

Baseline: Our baseline case for all experiments is the policy where we allocate equal numbers of cores to numbers of threads requested by the user. We set hard limits across VMs by using CPU resource limits, which prevents dynamic adjustment of under-utilized resources.

Default Manager (ESXi): The default manager allocates CPU resources based on the requested resource limits or reservations. Resource limits are hard constraints that can not be modified by the manager. On the other hand, resource reservations are soft constraints, such that the resources that are not utilized can be lent to other VMs by the ESXi manager. Therefore, we reserve n number of vCPUs for n number of threads requested for each VM. In this scenario, the default manager can lend any unused CPU resources to other VMs, but can never limit the VM usage.
Demand proportional: Demand metric for a workload has been defined as the maximum amount of utilization of the system for a given number of threads. Demand proportional policy distributes the available resources across VMs proportional to their demand estimations.
Scaling priority: Similar to the demand proportional approach, scaling priority approach uses the demand metric for all VMs to make resource distribution decisions. However, this technique favors the higher demand workloads (i.e., better scaling ones) over the lower demand ones, rather than proportionally distributing the available resources.
Proposed: The proposed technique incorporates both the scalability estimations through demand metric and the power efficiency through MHz/W metric. The proposed approach utilizes these two measurements to solve a maximization problem using linear programming, as explained in Section 3.

5.2. Placement Algorithms

We focus on three different placement techniques that are previously proposed to improve performance of consolidated environments, namely memory-based, similarity-based, and demand-based placement algorithms. The main idea behind all three placement algorithms is reducing the contention that might occur when consolidating multiple workloads, thus improving the performance. Each of these algorithms uses a different metric as a proxy to evaluate the potential contention. The goal is to create a balanced resource consumption across all VMs to improve the overall performance.

For evaluating the placement algorithms, we first collect the necessary measurements for each benchmark when they are executed alone and use offline data while making placement decisions. In Algorithm 3, we show the pseudo-code for the memory-based placement algorithm. Algorithm 3 first sorts all the benchmarks in the workload queue based on the last-level cache miss rates, which can be used as an indicator of the memory accesses. As a next step, Algorithm 3 starts grouping the benchmarks starting from the top of the list, then the bottom of the list, and then progresses through the list until the total number of threads or the total utilization of the benchmark group does not exceed predetermined threshold values.

Similarly, the demand-based algorithm applies the same idea using the `demand` metric, which is a virtualized environment specific metric. ESXi hypervisor provides VM-level metrics to pinpoint the CPU resource usage and bottlenecks. Resource demand of a VM can be estimated by adding these two metrics from `esxtop`.

`RUN`: The percentage of total schedule time of the VM, which excludes the system time (%`UTIL` = %`RUN` + %`SYS`).
`READY`: The percentage of time that the VM is ready to run, but not scheduled. This metric implies that the application will be able to utilize the CPU if more resources were allocated to the VM. Therefore, `READY` metric can be utilized to estimate maximum utilization level, which reflects the performance scalability characteristics of the applications.

By balancing the demand across the VM group, it is possible to reduce performance degradation due to CPU contention. These two placement techniques, memory and

ALGORITHM 3: Balanced Memory Placement

```
Input: $W_{ij}$ // Workload Matrix
Output: VM mapping
initialization;
k = 1;
sort($W_{ij}$.memaccess) // Sort based on memory accesses
for each VM in sorted.$W_{ij}$ ;
if $S_i.util < U_{max}$ and $S_i.thread < T_{max}$ then
    add sorted.$W_{ij}(k)$ to $S_i$;
    k = i − j − 1; // Reverse list index to continue from bottom
else
    k = k + 1; // Continue from the list
end
```

demand-based, focus on either memory or the CPU as the main source of contention. In order to be able to capture characteristics in other dimensions, similarity-based approaches are recently proposed [Delimitrou and Kozyrakis 2013]. Similarity-based techniques evaluate a set of performance counters and derive the similarities of benchmark on the selected dimensions through various correlation techniques. We use *Pearson's correlation* over the `USED%` of the applications similar to the technique described in prior work [Kim et al. 2013]. The final result of this evaluation is a score of similarity, which is then used as the main metric for sorting and grouping the applications to match them using a similar algorithm described in Algorithm 3.

5.3. Experimental Evaluation

In order to compare and evaluate the aforementioned resource allocation (RA) techniques, we create workload sets out of the benchmarks explained in Section 4. Each workload set consists of applications with various thread counts. We generate 100 workload sets such that each workload set consists of a total of 12 threads or 8 threads for AMD and Intel-based servers, respectively. Therefore, we aim to create a data center scenario in which the servers are not over-utilized. To fairly compare the resource distribution policies, we use the same default placement scenario for all policies. The default placement policy is a *first-fit bin packing* algorithm run on the list of applications with thread numbers requested.

In Figure 6, we present the performance improvement with various resource allocation techniques together with various placement techniques under 100W power cap for the AMD-based server. We compare the performance improvement with respect to the default baseline case, which uses the *first-fit* bin packing algorithm as the placement technique and with the hard CPU limits that are directly proportional to the number of threads requested. We also provide the breakdown of the total performance improvement as *placement* and *resource allocation* techniques. On the x-axis, we list the resource allocation techniques (grey) and the performance improvements with three different placement techniques (red, green, purple). Similarly, Figure 7 shows the results for the same set of experiments with a power cap of 130W. We choose 130W as a representative value for the medium power cap and 100W as a representative value for the high power cap, meaning low power consumption. AMD-based server consumes 65W in *idle* mode with a peak power value of 171W.

The first observation from these figures is that incorporating the power weights of the application in the LP solution improves the performance by up to 10% from the best performing *placement+RA* technique, which uses only the performance scalability information for resource allocation with similarity-based placement (x-axis: Scaling Prop.

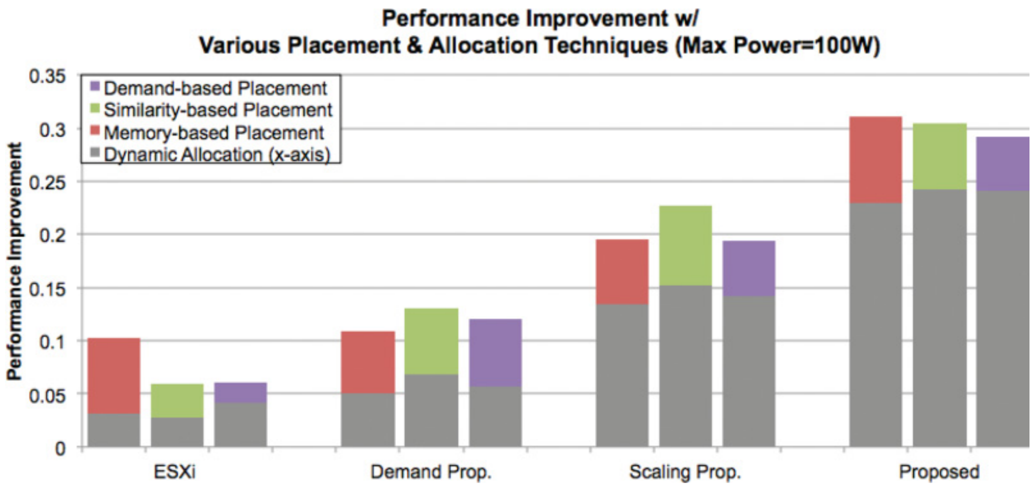

Fig. 6. Performance improvements with various resource allocations (x-axis) together with various placement techniques (red, green, and purple bars) under a power cap of 100W. The proposed technique improves the performance by up to 21% with respect to the default ESXi manager.

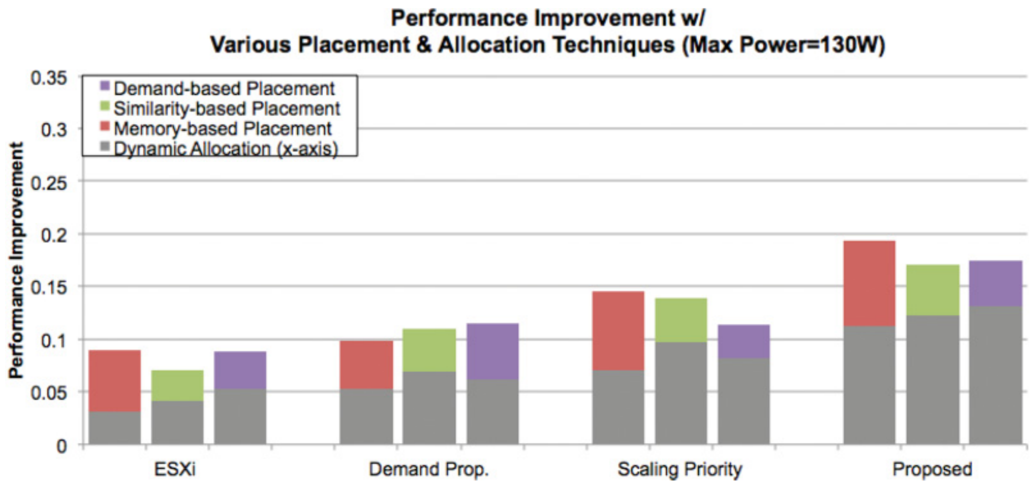

Fig. 7. Performance improvements with various resource allocations (x-axis) together with various placement techniques (red, green, and purple bars) under a power cap of 130W. The proposed technique improves the performance by up to 11% with respect to the default ESXi manager.

with grey and green bars). For the scaling-based resource allocation, similarity-based placement works best; as for the similarity weights, scaling information has the highest impact. However, for higher power ranges (Figure 6), as power weight information becomes more predominant, the similarity-based approach starts to favor scalability information in a lesser degree, which causes imperfect matches from the scalability perspective. For the proposed approach, which incorporates the scaling and power weight information, memory-based placement provides the highest improvements. The underlying reason is that the similarity-based approach fails to capture the importance of the power weight information and highly favors the scalability information. As expected, favoring scalability works better with the scalability-based resource allocation

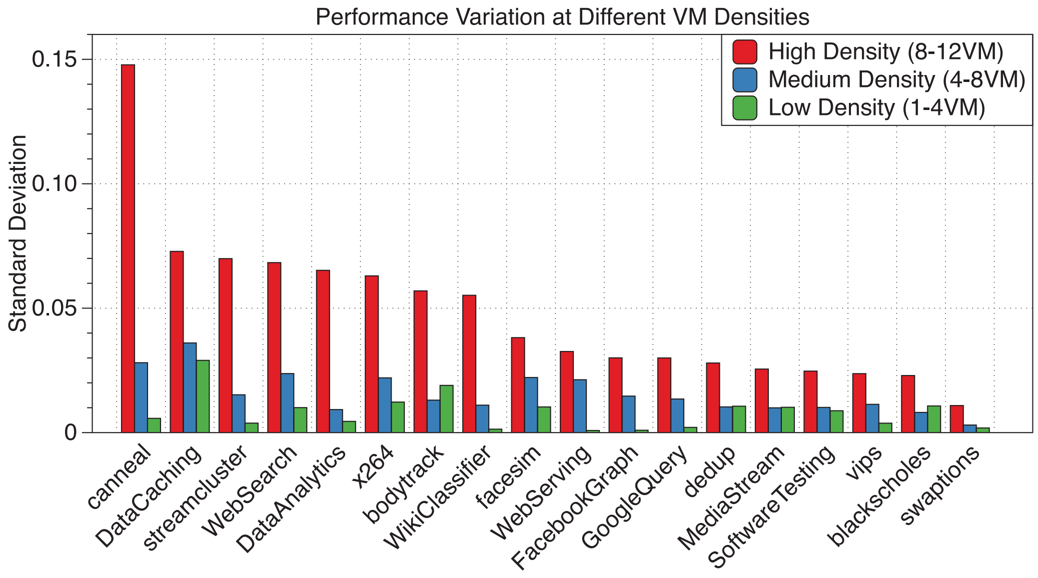

Fig. 8. Performance variation for all applications for various VM density cases. Higher VM density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity.

approach. For the proposed approach, memory-based placement improves the performance by up to 3.5% with respect to the best performing placement algorithm.

The second observation is that the performance improvements due to resource allocation techniques are consistently higher than the performance improvements due to placement techniques, with only a few exceptions. Furthermore, resource allocation-based improvements become significantly higher than placement-based improvements for lower peak powers (i.e., higher power cap). The reason is that for tight power budgets, power and scaling variations across applications become more apparent, while the high power budgets provide enough room to consolidate all applications with minimal restrictions. As we evaluate a scenario where there is no oversubscription, high power budgets converge to the default performance without needing a complicated resource allocation techniques, and the benefits due to placement techniques become more dominant. This also explains the reason why resource allocation techniques bring more benefits at lower power budgets.

5.4. The Impact of VM Density on Placement Techniques

In order to evaluate the impact of VM density of the benefits due to placement techniques, we create three workload set scenarios with various average utilization values. Low Load represents a case where the majority of the workloads are utilizing the system around 20%. We call this case the Low Load, since the average utilization of the server will be low when there is no consolidation. Consolidating a Low Load workload set leads to higher numbers of VMs to be consolidated. This is expected to have implications when choosing a placement algorithm.

The benefit of choosing a good placement algorithm is due to reducing the potential resource interference across multiple applications (or VMs). In Figure 8, we show the performance degradation due to consolidation for various placement algorithms at three different VM density scenarios. High VM Density represents the case that has the highest number of VMs consolidated at a particular time period. By using the Low

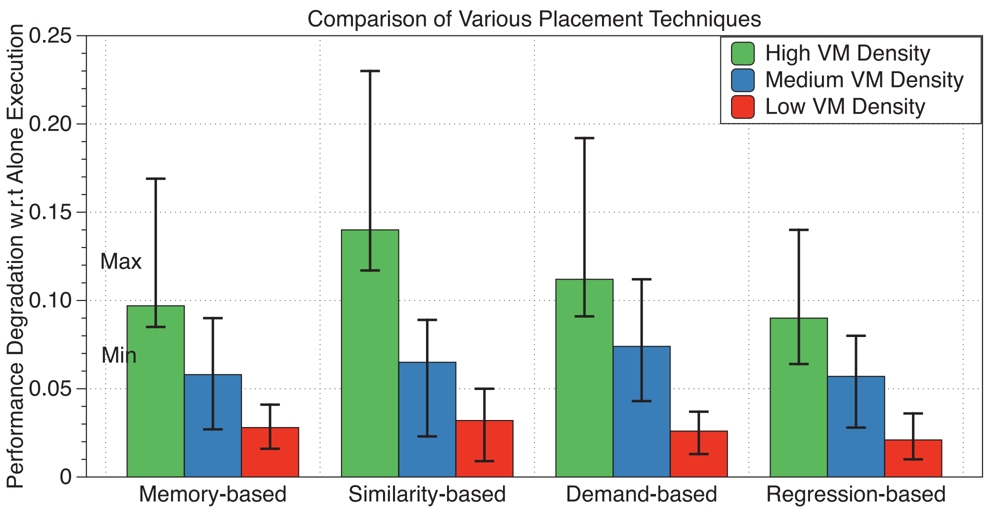

Fig. 9. Comparison of placement techniques in terms of performance degradation (i.e., lower is better).

Load workload set, we can create consolidation cases where a higher number of VMs are consolidated at the same time. Similarly, by using the High Load set, we end up with consolidation cases with low VM densities. As the placement algorithms become more critical when there is more contention, the High VM Density case is expected to have the most benefits from placement algorithms.

In Figure 9, we show the performance degradation at different VM densities for 4 different placement algorithms. In addition to the placement algorithms described in Section 5.2, we also evaluate a *regression-based* placement algorithm, which is a simple linear regression model that uses an offline data to predict the best possible VM matching for a given set of workloads. The offline data includes the performance results of each benchmark when consolidated with others for all workload sets. Therefore, the *regression-based* approach is expected to be the best possible placement algorithm by using the offline data available. Depending on the placement algorithm, the degradation ranges from 24% to 1%, where the High VM Density case causes the highest degradation. For Medium and Low VM densities, there is minimal differences across different placement algorithms. However, at High VM densities, choosing a memory-based placement algorithm reduces the degradation by up to 11%. For higher VM densities, the memory overhead for VM creation causes additional memory contention. Therefore, the memory-sensitive approach can bring up to 7% with respect to other placement techniques.

In order to look at the impact of increased VM density and the higher memory stress, we compared three placement techniques for the same 100 consolidation sets and reported the best performing placement techniques for each of these distinct workload sets. In Figure 10, we color code the best performing placement technique for varying VM densities and active memory sizes. As the figure shows, memory-based placement needs to be favored for high memory and high VM density consolidation scenarios, due to the aforementioned reasons. On the other hand, the similarity-based placement technique starts to perform better for lower VM density and lower memory sizes, as the CPU resources become more critical for CPU-heavy workload sets and similarity-based favors the CPU resource demand metric when making placement decisions, as also explained in Section 5.2.

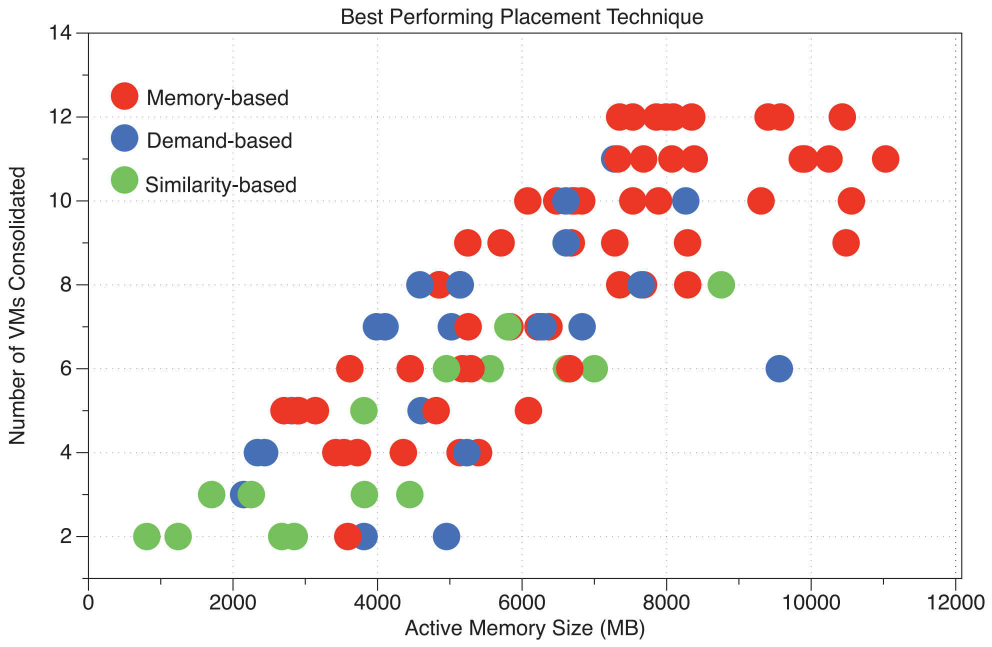

Fig. 10. Best performing placement technique for various VM density and active memory size. The memory-based technique is superior to other techniques with increasing number of VMs consolidated at the same time, which also leads to higher active memory size.

The main limitations of the proposed technique is as follows: **Scale & Cap** achieves significant performance improvements under power constraints by exploiting the performance and power scalability differences across consolidated applications. Therefore, application sets that exhibit similar power and performance behavior would not be ideal candidates for the proposed technique. For instance, consolidating multiple instances of a specific application will not provide any power and/or performance improvements. In our experiments, we evaluate mainly CPU and memory-bounded applications. In order to achieve similar performance benefits for workloads that are disk, or network-bounded, additional metrics and monitoring capabilities might be required. Although **Scale & Cap** can be implemented on servers without any power monitoring sensors, the implementation would be more complex and might require offline data collection to achieve the highest benefits. Therefore, **Scale & Cap** works best with servers that have built-in power monitoring capabilities.

6. RELATED WORK

VM management is a multi-dimensional problem, where the objective is optimally distributing the limited resources to meet competing objectives, such as power and performance. Therefore, there is a large body of work in VM management, each of which aims to provide solutions for various execution scenarios from various stand points. In this work, we look at the VM management problem from the resource distribution/allocation perspective. However, in this section, we summarize the prior work in VM management that is relevant to our work. We mainly focus on three main categories, namely, placement and scheduling techniques, resource allocation techniques, and power-capping techniques.

6.1. Placement and Scheduling

Placement techniques mainly target to reduce the performance degradation due to interference across applications and VMs [Beloglazov et al. 2012]. Therefore, the maximum performance improvement due to placement algorithms is limited with the maximum performance degradation due to interference. Placement techniques aim to find the best application groups that will create the least contention due to interference. One of the main sources of contention is the rate of cache accesses. For instance, co-locating (i.e., placing) multiple memory-bound applications together will degrade the performance due to increased amount of cache contention.

In order to reduce the degradation due to co-location, many techniques focus on balancing the memory accesses across application groups [Dhiman and Rosing 2007]. For instance, co-locating memory-intensive applications with CPU intensive applications will reduce the cache contention when compared to the case where memory-intensive ones are co-located together. Another line of work used various metrics to capture the interference impact at various dimensions [Delimitrou and Kozyrakis 2013; Jiang et al. 2010; Zhang et al. 2014]. Although these techniques are reported to significantly improve the performance of co-located applications, they are agnostic about the power constraints that might be enforced due to a variety of reasons. Therefore, placement techniques are limited in terms of having the capability of managing multiple objectives such as power and performance. Resource allocation techniques are essential to be able to control multiple objectives at the server level.

The common goal of resource allocation techniques is improving the performance of the server through efficiently distributing the available resources (i.e., compute and/or power resources) [von Laszewski et al. 2009; Shafique et al. 2013]. One line of work in resource allocation considers only the performance as the single dimension to optimize.

6.2. Resource Allocation

Consolidation and migration policies target balancing the activity on various server components such as CPU, memory, or disk to improve energy efficiency (e.g., Merkel et al. [2010]). Modern virtualization environments such as Xen, KVM, and vSphere provide resource management mechanisms to improve the efficiency of the server nodes mainly through scheduling techniques [Xen 2011; VMware 2013; KVM 2015]. While KVM relies on default Linux resource management, Xen and vSphere provide management options such as Xen Management Tools and vSphere's Distributed Resource Scheduler (DRS), which mostly rely on VM migration to provide balanced load distribution across server nodes. Merkel et al. [2010] propose co-scheduling and migration policies based on task activity vectors, which are used to characterize applications. Their proposed policies mitigate the resource contention through vector balancing for single-threaded applications. Kusic et al. [2008] propose a dynamic resource provisioning framework based on look-ahead control for virtualized server environments. Vasic et al. [2012] propose the *DejaVu* framework that makes resource allocation decisions based on the history of the VMs to reduce the resource management overhead. In order to improve the clusters with workloads that heavily utilize the disk, Romosan et al. [2005] propose co-scheduling algorithms based on load balancing frequently used files. Zheng et al. [2009] present an empirical infrastructure for data center management. Their proposed infrastructure allocates a server node (i.e., sandbox) to experimentally derive the energy/performance tradeoffs. Beloglazov and Buyya [2010] propose a threshold-based dynamic consolidation technique, which provides SLA performance guarantees. The proposed algorithm selects VMs to migrate to different physical nodes based on resource utilization [Beloglazov and Buyya 2010]. Bonvin et al. [2011] propose a dynamic resource allocation algorithm to meet SLA performance and availability guarantees by adding or removing new resources (i.e., allocation of cores or entire new server nodes). However, their proposed work does not consider power and energy aspects. Wang et al. [2012] propose a framework that allows user-specified workload provisioning policies to optimize energy efficiency on clusters. Their framework allocates threads to available cores across the cluster depending on the user-specified performance/power constraints. Vasic et al. [2012] propose the *DejaVu* framework that makes resource allocation decisions based on the history of the VMs to reduce the resource management overhead.

6.3. Power Capping

Most of the modern processor cores support dynamic voltage-frequency scaling (DVFS) and power gating capabilities. Therefore, DVFS and core power gating have become traditional power management knobs [Li and Martinez 2006]. Recent commercial servers also provide power-capping capabilities [Samson 2009]. For example, Intel Sandybridge provides a power estimator and a runtime average power limiter (RAPL) [David et al. 2010].

Raghavendra et al. [2008] propose a global power management technique for clusters to coordinate the power provisioning for individual nodes under power constraints [Raghavendra et al. 2008]. Reda et al. [2012] propose a runtime controller to meet the peak power constraints through DVFS and packing threads onto a smaller number of cores, while optimizing the application performance. Ma et al. [2012] propose a power-capping technique by power-gating the cores and applying per-core DVFS for a mixture of single and multi-threaded applications running on native servers.

For capping the power in virtualized environments, Nathuji and Schwan [2008] design a power allocation technique for VMs to improve the performance for a given power budget by allocating power budgets proportionally across VMs according to the service level agreement (SLA) requirements of individual VMs. The proposed technique uses CPU utilization data to distribute the available power budget. Dhiman et al. [2009] propose a VM scheduling technique that estimates VM-level CPU and memory usage based on system-level metrics to guide co-scheduling and migration decisions. Their proposed technique consolidates the applications that have complementary resource usage characteristics to reduce the performance degradation. Hwang et al. [2012] study the impact of CPU consolidation in virtualized multi-core environments. Their study investigates finding the optimum VM density for multi-core processors for single-threaded applications that have distinct characteristics (i.e., memory/CPU-bounded) and they propose a consolidation policy that uses DVFS and core power gating. Vasic et al. [2012] introduce the *DejaVu* framework that makes resource allocation decisions based on the history of the VMs to reduce the resource management overhead. Another line of work in VM resource management targets reducing the resource contention to improve the efficiency of the virtualized servers [Delimitrou and Kozyrakis 2013; Kim et al. 2013]. Hankendi et al. [2013] proposed a dynamic power-capping technique for consolidated environments that uses performance scalability information to make resource allocation decisions, while meeting power caps.

Scale & Cap considers multi-threaded application-specific aspects while making resource distribution decisions. The main distinguishing aspects of **Scale & Cap** are as follows:

—**Scale & Cap** accurately captures the scalability characteristics of multi-threaded applications in multiple dimensions (i.e., power and performance) to make resource allocation decisions across a various number of VMs, while prior work can not capture both power and performance scalability aspects.

—We implement and evaluate our technique on a real-life server and show that our resource allocation technique can work seamlessly with placement techniques to further improve the energy efficiency, while prior work focuses on either placement or allocation techniques without evaluating the interactions across different techniques.
—We provide analyses on the interactions between placement and resource allocation techniques and provide guidelines for developing better management techniques under various execution scenarios (i.e., memory-bound execution scenarios, high/low VM density scenarios).
—**Scale & Cap** can handle managing up to 12 consolidated VMs at a time for a 12-core system, while prior work evaluates either a constant number of VMs or low VM density cases.

7. CONCLUSION

Energy-related costs are among the biggest contributors to the total cost of ownership of the data centers. Thus, constraining the peak power consumption has become a common practice for cost management and reliable power delivery. As more than 50% of the cloud resources are virtualized, it becomes essential to design power-capping solutions for virtualized servers. In tandem, multi-threaded applications start to emerge on the cloud resources from various application domains. Multi-threaded applications introduce additional challenges due to their more complex characteristics such as performance scalability.

In this work, we present **Scale & Cap**, a resource allocation technique that incorporates the multi-threaded specific performance scalability and power efficiency characteristics to distribute the available resources across multiple VMs running heterogeneous applications. We formulate our solution as a linear programming-based algorithm and implement **Scale & Cap** on two multi-core servers. We evaluate various resource allocation and placement techniques together and provide insights regarding the interaction between placement and resource allocation techniques. Our results also show that for tight power budgets, resource allocation brings up to 21% performance improvements in comparison to only using a placement algorithm.

REFERENCES

Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computing Systems* 28, 5 (2012), 755–768.

Anton Beloglazov and Rajkumar Buyya. 2010. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *International Workshop on Middleware for Grids, Clouds and e-Science*. 1–6.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Nicolas Bonvin, Thanasis G. Papaioannou, and Karl Aberer. 2011. Autonomic SLA-driven provisioning for cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid)*. 434–443.

Lucinda Borovick. 2011. The benefits of a virtualized approach to advanced-level network services. *International Data Corporation (IDC), Whitepaper* (February 2011).

Hao Chen, Can Hankendi, Michael C. Caramanis, and Ayse K. Coskun. 2013. Dynamic server power capping for enabling data center participation in power markets. In *International Conference on Computer-Aided Design (ICCAD'13)*. 122–129.

Ryan Cochran, Can Hankendi, Ayse Kivilcim Coskun, and Sherief Reda. 2011. Pack & cap: Adaptive DVFS and thread packing under power caps. In *International Symposium on Microarchitecture (MICRO)*. 175–185.

Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory power estimation and capping. In *International Symposium on Low Power Electronics and Design (ISLPED)*. 189–194.

Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 77–88.

Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. 2009. vGreen: A system for energy-efficient computing in virtualized environments. In *International Symposium on Low Power Electronics and Design (ISLPED)*. 243–248.

Gaurav Dhiman and Tajana Simunic Rosing. 2007. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *International Symposium on Low PowerElectronics and Design (ISLPED)*.

Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 37–48.

Can Hankendi and Ayse K. Coskun. 2012. Reducing the energy cost of computing through efficient co-scheduling of parallel workloads. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 994–999.

Can Hankendi, Sherief Reda, and Ayse K. Coskun. 2013. vCap: Adaptive power capping for virtualized servers. In *International Symposium on Low Power Electronics and Design (ISLPED)*. 415–420.

Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 199–212.

Inkwon Hwang, Timothy Kam, and Massoud Pedram. 2012. A study of the effectiveness of CPU consolidation in a virtualized multi-core server system. In *International Symposium on Low PowerElectronics and Design (ISLPED)*. 339–344.

Canturk Isci, James E. Hanson, Ian. Whalley, Malgorzata Steinder, and Jeffrey O. Kephart. 2010. Runtime demand estimation for effective dynamic resource management. In *Network Operations and Management Symposium (NOMS)*. 381–388.

Congfeng Jiang, Jilin Zhang, Jian Wan, Xianghua Xu, Yuyu Yin, Ritai Yu, and Changping Lv. 2010. Power aware resource allocation in virtualized environments through VM behavior identification. In *2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM'10)*. 313–318.

Jungsoo Kim, Martino Ruggiero, David Atienza, and Marcel Lederberger. 2013. Correlation-aware virtual machine allocation for energy-efficient datacenters. In *Conference on Design, Automation and Test in Europe (DATE)*. 1345–1350.

Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang. 2008. Power and performance management of virtualized computing environments via lookahead control. In *International Conference on Autonomic Computing (ICAC)*. 3–12.

KVM. 2015. Kernel-based Virtual Machine. Retrieved from http://www.linux-kvm.org/page/FAQ.

Jian Li and Jose F. Martinez. 2006. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *International Symposium on High-Performance Computer Architecture (ISCA)*. 77–87.

Kai Ma and Xiaorui Wang. 2012. PGCapping: Exploiting power gating for power capping and core lifetime balancing in CMPs. In *Parallel Architecture and Compilation Techniques (PACT)*. 13–22.

Andreas Merkel, Jan Stoess, and Frank Bellosa. 2010. Resource-conscious scheduling for energy efficiency on multicore processors. In *European Conference on Computer Systems (EuroSys)*. 153–166.

Ripal Nathuji and Karsten Schwan. 2008. VPM tokens: Virtual machine-aware power budgeting in datacenters. In *International Symposium on High Performance Distributed Computing (HPDC)*. 119–128.

Ripal Nathuji, Karsten Schwan, Ankit Somani, and Yogendra Joshi. 2009. VPM tokens: Virtual machine-aware power budgeting in datacenters. *Cluster Computing* (2009), 189–203.

Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. 2008. No "power" struggles: Coordinated multi-level power management for the data center. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 48–59.

Sherief Reda, Ryan Cochran, and Ayse K. Coskun. 2012. Adaptive power capping for servers with multi-threaded workloads. *IEEE Micro* 32, 5 (2012), 64–75.

Ru Romosan, Doron Rotem, Arie Shoshani, and Derek Wright. 2005. Co-scheduling of computation and data on computer clusters. In *17th International Conference on Scientific and Statistical Database Management (SSDBM)*. 103–112.

Ted Samson. 2009. AMD Brings Power Capping to New 45nm Opteron Line. Retrieved from http://www.infoworld.com/d/green-it/amd-brings-power-capping-new-45nm-opteron-line-906.

Muhammad Shafique, Benjamin Vogel, and Jorg Henkel. 2013. Self-adaptive hybrid dynamic power management for many-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 51–56.

Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. 2012. DejaVu: Accelerating resource allocation in virtualized environments. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 423–436.

VMware. 2013. Resource Management with VMware DRS. Retrieved from http://www.vmware.com/pdf/vmware_drs_wp.pdf.

VMware. 2015. vSphere SDK for Perl Documentation. Retrieved from https://www.vmware.com/support/developer/viperltoolkit/.

G. von Laszewski, Lizhe Wang, A. J. Younge, and Xi He. 2009. Power-aware scheduling of virtual machines in DVFS-enabled clusters. In *Cluster Computing and Workshops (CLUSTER)*. 1–10.

Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, K. Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA)*. 488–499.

Wei Wang, Tanima Dey, Ryan W. Moore, Mahmut Aktasoglu, Bruce R. Childers, Jack W. Davidson, Mary Jane Irwin, Mahmut Kandemir, and Mary Lou Soffa. 2012. REEact: A customizable virtual execution manager for multicore platforms. In *Virtual Execution Environments*. 27–38.

Xen. 2011. Xen Management Tools. Retrieved from http://wiki.xen.org/wiki/Xen_Management_Tools.

Yunqi Zhang, M. A. Laurenzano, Jason Mars, and Lingjia Tang. 2014. SMiTe: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. In *International Symposium on Microarchitecture (MICRO)*. 406–418.

Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. 2009. JustRunIt: Experiment-based management of virtualized data centers. In *USENIX Annual Technical Conference*. 18–18.

Received July 2015; revised May 2016; accepted August 2016