Model-driven Consolidation of Java Workloads on Multicores

***Abstract*—Optimal resource allocation and application consolidation on modern multicore systems that host multiple applications is not easy. Striking a balance among conflicting targets such as maximizing system throughput and system utilization while minimizing application response times is a quandary for system administrators. The purpose of this work is to offer a methodology that can automate the difficult process of identifying how to best consolidate workloads in a multicore environment. We develop a simple approach that treats the hardware and the operating system as a black box and uses measurements to profile the application resource demands. The demands become input to a queueing network model that successfully predicts application scalability and that captures the performance impact of consolidated applications on shared on-chip and off-chip resources. Extensive analysis with the widely used DaCapo Java benchmarks on an IBM Power 7 system illustrates the model's ability to accurately predict the system's optimal application mix.**

Keywords: **performance models, queueing networks, application consolidation, Java, multicores**
Submission Category: **full paper, PDS track**
Word count: **approximately 9,300**
We declare that the material has been cleared through the authors' affiliations.

I. Introduction

Multicore architectures have become the standard both for general purpose PCs and for high-end servers. Such platforms provide an environment that invites consolidating several applications within the same processor to fully take advantage of the available parallelism. As multicore processors become basic building blocks of data centers and cloud computing platforms, the problem of effective consolidation becomes very challenging as interference between collocated workloads[1] may significantly affect their performance [12], [19]. Server virtualization has offered solutions to the problems of software heterogeneity and fault isolation but not to the problem of performance isolation. Consequently, there is a lot of research on resource partitioning that aims at performance isolation on multicores [6], [11], [27]. Resource partitioning has been shown to be costly and may even result in inefficient resource usage [27], defeating the purpose of consolidation for reducing infrastructure and electricity costs. As such, consolidating applications is often conflicting with performance isolation, especially since performance degradation due to collocation may be difficult to characterize and foresee.

In this paper, we focus on providing a systematic methodology that can provide answers to two important questions: first, what is the system and application performance when multiple applications execute on multicore systems, and second what is the best way to collocate applications, i.e., what are the workload characteristics of competing applications that are best to be matched in order to obtain an optimal workload "mix" such that performance interference on the multicore system is minimized. Here, we focus on the optimization of the ratio of two conflicting measures: maximize system throughput (i.e., increase consolidation by raising the number of applications that are collocated) while keeping the application's end-to-end times as low as possible (i.e., by minimizing their performance interference). The ideal workload mix should maximize the ratio of system throughput versus the average application end-to-end response time.

To find the optimal workload mix, one could use exhaustive experimentation, i.e., execute all possible different combinations of collocated workloads for various system loads. This approach can provide the optimal mix given a performance target but is largely non-viable. Workloads tend to be dynamic so it is imperative for smooth system operation to provide a methodology that can provide on-the-fly—as the workload changes—the optimal (or a close to optimal) mix rather than relying on "after-the-fact" results or trying out all possible combinations. Here, we advocate the use of a surprisingly simple closed queueing network model that is able to accurately capture contention among applications in a multicore system for common on-chip and off-chip resources and that can effectively propose the optimal workload mix.

Closed queueing networks [16] have been successfully applied to model various computing and software systems, including complex multi-tier applications that operate under bursty workloads [18], [28], [29]. Most importantly, it has been shown that complex interactions among software and hardware (e.g., caching or locking effects) can be well captured by careful parameterization [18]. The abstraction provided by queueing networks, although surprisingly simple, can provide a good solution. Correct model parameterization is key to capturing resource interference effects among competing applications. In addition, model parameterization depends on effective characterization of the dynamic behavior of applications.

[1] In this paper we use the terms application, job, workload, and JVM interchangeably.

The first contribution of this paper is the development of a simple black box methodology for application profiling that is light-weight and based on standard resource monitoring tools. We illustrate that this simple profiling provides measurements that are easy to obtain but are also accurate enough to capture performance interferences of the various resources. From these measurements we calculate the per-application *service demands* for each basic system resource, i.e., CPU and disk. The second contribution of this work demonstrates how to use a simple queueing network to predict the scalability of single and multiple instances of applications on multicores and, most importantly, suggest an optimal application mix for a target system operational point. The queueing network is parameterized using the service demands measured by the proposed black box profiler.

The target applications that we consider here are written in Java, an object-oriented, managed language. The platform used is an IBM Power 7 system. Application profiling is first illustrated on simple Java microbenchmarks and also on fully-fledged applications from the DaCapo benchmark suite [3]. This suite includes a wide variety of common real-world Java applications with complex runtime behavior and several execution phases. In addition, this benchmark collection represents a wide variety of applications.

In our evaluation, each Java application is run in a loop by a separate Java Virtual Machine (JVM) process, resulting in a complex overall system with many layers: hardware (multicores with complex data paths and memory hierarchies), operating system (thread scheduling and memory management), and virtual execution environment (providing just-in-time compilation and automated memory management). Our black box profiler is not Java-specific and can profile any workload. We evaluate our approach with Java workloads because they are increasingly important in server environments and because it is particularly difficult to predict the impact of garbage collection and just-in-time compilation on runtime performance.

For each DaCapo application, we illustrate the effectiveness of black box profiler to gather one dynamic metric—*service demand*—for each resource used. We demonstrate that this metric is sufficient for defining the optimal application mix, i.e., how many instances of each application to deploy on a given machine to simultaneously maximize throughput and minimize end-to-end execution time. Using the proposed model, one can easily use the underlying architecture at its best, without having in-depth knowledge on architecture details. Extensive experiments on an IBM Power 7 system illustrate the model's ability to reflect well architectural components within the resource demand metric and to offer answers to the following difficult questions: how many applications should one consolidate on the same system and which applications are better to collocate so as to maximize certain performance targets.

This paper is organized as follows. A cautionary example of projecting performance interference with back-of-the-envelope calculations is given in Section II. The model of consolidating an optimal application mix is explained in Section III. The proposed application profiling methodology for single and multicore systems is described in Section IV. Section V contains the experimental results. Related work is presented in Section VI. Section VII concludes this paper.

II. Projecting Performance Interference: A Cautionary Example

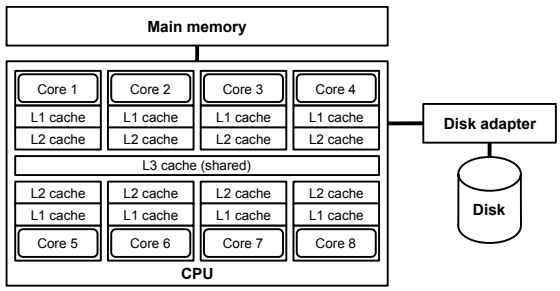


Figure 1. High-level view of the target hardware platform.

In this section we illustrate the difficulty of projecting performance interference in a multicore system. The target software is a selection of Java benchmarks from the DaCapo benchmark suite (dacapo-9.12-bach release) [3] as representative of contemporary workloads. Java applications are usually compiled to an intermediate code representation and executed in a virtual machine that provides a platform-independent abstraction of the underlying system. This approach eases code portability and allows many runtime optimizations, such as Just-In-Time (JIT) compilation and automatic memory management performed by the Garbage Collector (GC). However, the presence of a virtual machine layer in the software stack makes more difficult to understand and predict performance of concurrently executing Java applications on modern hardware platforms.

The target hardware is an IBM Power 750 Express server, from which we use a logic partition on a single processor board hosting 8 cores running at 3.00 GHz and 64 GB of RAM. The disk adapter is a PCI-X 266 Planar 3Gb SAS, and the disk is a Hitachi Ultrastar C10K300, 147 GB, 10000 RPM, with 64 MB buffer. The system runs AIX 6.1 (64 bit) and we use IBM J9 JVM SR8-FP1 (64 bit) in server mode, with 2 GB heap size and default GC algorithm. This configuration is commonly found in many server environments. Figure 1 provides a high-level view of the hardware platform. The performance of application consolidation here is difficult to predict because each layer offers different optimization features that may impact key performance factors.

To understand and predict runtime performance of consolidated Java applications, a wide range of system resource

Table I
METRICS FOR THREE BENCHMARKS FROM THE DACAPO 9.12 SUITE.

(a) CPU-related metrics
statistics collected with hpmstat (first 4 columns) and with iostat (last 4 columns)

benchmark	completed inst.	consumed cycles	instr./ cycles	MIPS	utilization [%]	user mode [%]	system mode [%]	idle [%]
batik	8.65E+11	8.01E+11	1.08	450.33	27.6	27.1	0.5	72.4
fop	7.27E+11	9.30E+11	0.78	378.39	32.1	31.4	0.7	67.9
luindex	5.15E+11	4.42E+11	1.16	268.12	15.2	11.4	3.8	84.8

(b) Disk-related metrics
statistics collected with iostat -D [hdName]

benchmark	actitity time [%]	accessed KB/s	operations/ second	avg. service time [ms]	avg. queueing time [ms]	avg. service queue size
batik	5.5	34.0	8.9	7.9	1.0	0
fop	10.4	160.2	18.0	6.4	0.4	0
luindex	55.7	618.8	155.1	13.1	9.1	2

(c) Memory-related metrics
statistics collected with benchmark harness instrumentation (first 3 columns) and hpmstat (last 3 columns)

benchmark	allocated heap mem. [MB]	allocated non_heap mem. [MB]	total allocated mem. [MB]	accesses to local mem.	memory load traffic [MB/s]
batik	90.49	21.32	111.81	7.22E+07	4699.92
fop	92.57	20.93	113.50	1.96E+08	12765.87
luindex	9.59	15.04	24.63	1.90E+07	1233.88

(d) Cache-related metrics
statistics collected with hpmstat

benchmark	L3 cache hit rate [%]	L3 prefetch hit rate [%]	castouts to memory	CPU stalls	stalls/cycles
batik	87.59	82.74	1.71E+08	2.08E+11	0.26
fop	89.16	85.03	4.92E+08	3.23E+11	0.35
luindex	93.17	93.72	4.12E+08	1.25E+11	0.28

statistics (e.g., CPU, memory, and disk[2]) are collected as input for solving a multi-dimensional "bin-packing" problem. Typically, for each system component, one can collect statistics at different layers, i.e., at the JVM level, at the Operating System (OS) level, and at the hardware (HW) level. JVM-level statistics provide detailed performance of the application, while OS- and HW-level statistics reflect the aggregate behavior of multiple instances of applications and include, for instance, scheduling overheads.

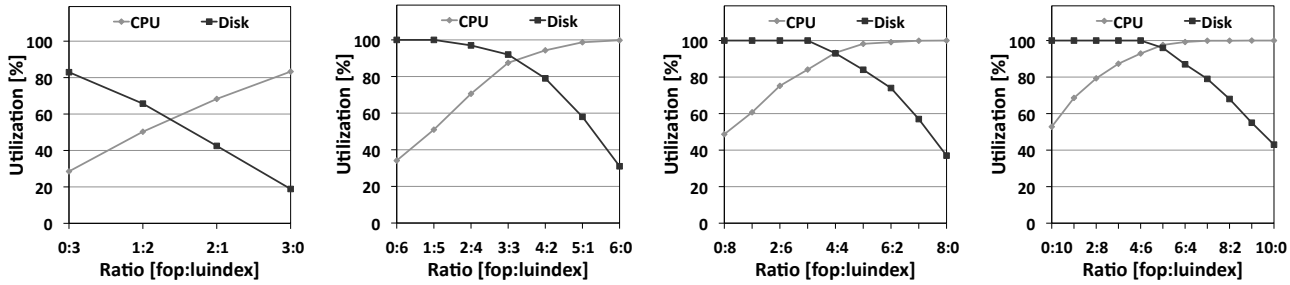
We now provide an example of the metrics that can be computed for different resources and we further show that a naïve analysis may lead to wrong conclusions. We use the `iostat` tool to collect statistics about CPU usage at the OS level and disk-related metrics at the HW level. In addition, we collect detailed statistics about CPU and memory usage via the `hpmstat` tool, which allows the collection of metrics over a custom observation period. Warm up time is 2 minutes (where measurements are ignored) and we observe the system for the next 3 minutes.

Table I shows four sets of metrics collected from a selection of benchmarks of the DaCapo 9.12 suite[3] : (a) CPU, (b) disk, (c) memory, and (d) cache-related statistics. For each benchmark, results are collected by executing multiple iterations of the same benchmark within the same JVM process, using the default workload size. The combination of CPU utilization in different modes, such as user, system, and idle time, as well as various disk operations (e.g., average service and queueing time), can characterize a wide range of CPU- and IO-intensive workloads [17], [26]. For large multicore systems, the performance interference on shared resources (e.g., memory and cache) becomes more apparent when collocating applications [6], [11].

Table I clearly highlights the difficulty of the problem: from the measurements shown, it is unclear how to estimate application scalability when the number of cores increases as it is also difficult to predict performance interference among different Java applications. The CPU and IO utilization metrics (see the "utilization" column in Tables I(a) and I(b)) appear to be a good overall measure that one could use to gauge how many Java applications could be executed simultaneously without significant performance interference. We focus on *fop* and *luindex*, as the first one is CPU intensive and the second one is IO intensive. These two applications are good to execute simultaneously in the system because it is possible to interleave the usage of the two resources among the consolidated applications. In addition, trying

[2]In this exposition, we do not focus on applications that make intensive use of the network, therefore we do not collect any statistics at the network level.

[3]Due to lack of space, usage statistics are presented here for a few representative benchmarks only.



(a) 3 JVMs (b) 6 JVMs (c) 8 JVMs (d) 10 JVMs

Figure 2. CPU and disk utilization of different mixes of *fop* and *luindex*.

to balance the utilizations across different resources helps eliminate a single bottleneck resource and maximize overall system throughput [21]. CPU utilization for *fop* is measured at 32.1%, while for *luindex* it is only 15.2%. Looking at the "disk activity time" in Table I(b), *luindex* seems to be 5.36 times more disk intensive than *fop*. Based on a simple back-of-envelope calculation, one could project that executing 2 instances of *fop* and a single instance of *luindex* would result roughly in CPU and disk utilizations being 79.4% and 76.5%, with CPU and disk nearly equally utilized without a persistent bottleneck. With this logic, executing more than two instances of *fop* and one of *luindex* projects resource utilizations as high as 100%, which would greatly deteriorate per-application end-to-end times.

To validate if this simple utilization rule is correct, we conduct exhaustive experiments of all possible combinations of *fop* and *luindex*, with a total number of simultaneously executed applications being 3, 6, 8, and 10. Figure 2 illustrates the overall CPU and disk utilization for all possible application mixes. In particular, Figure 2(a) shows that when we execute two instances of *fop* and one instance of *luindex* simultaneously, then the CPU and disk have utilizations equal to 68.3% and 42.5% respectively, far away from our projected values. This implies that we can indeed increase the number of simultaneously executing JVMs, which is shown in Figures 2(b) to 2(d). The figures clearly show that the ratios maximizing utilization of both resources are 3:3, 4:4, and 5:5, for 6, 8, and 10 JVMs respectively, while both of CPU and disk approach 100% utilization for as high as 10 JVMs[4]. This example obviates the need for a model that captures performance interference among applications and that can use aggregate information from measurements to provide reliable predictions.

III. A Model for Consolidating the Optimal Application Mix

The previous section highlights the need for a model to consolidate applications in a system. System resources should be utilized to maximize system throughput while maintaining a relatively fixed operational cost *and* the performance of individual applications should not suffer significantly because of the increased number of applications that execute simultaneously. Focusing on two conflicting performance metrics, i.e., maximizing throughput while minimizing application end-to-end time, we aim at maximizing *power* Φ that is defined as the ratio of normalized throughput X' over average job response time[5] T [16]:

$$\Phi = \frac{X'}{T}, \tag{1}$$

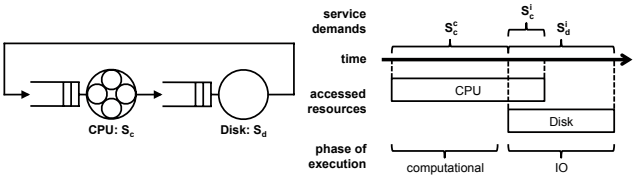
where X' is the system throughput weighted by service demands on different resources. The exact formula of X' is listed in [21]. Note that the definition of power Φ given here is clearly unrelated to the metric of electrical power consumption in the system.

Intuitively, to maximize throughput in a system with multiple applications that compete for multiple resources, the utilization levels of the various resources should be equal or close to equal. As a result, no clear resource becomes the system bottleneck where queues build up. Rosti et. al [21] have shown that aiming at an equal utilization point of all system resources optimizes Φ.

As illustrated in Section II, we can crudely classify applications as CPU or IO intensive. To capture the effect of CPU or IO demand on application response time, we can model a multicore system executing multiple applications as a two-station closed queueing network, see Figure 3(a). The first station represents the aggregate computational capacity from available cores in the system (ranging from 1 to 4 in our experiments) and the second station corresponds to the disk. The cumulative time an application spends on the CPU station includes the memory and cache accessing times. Yet, although the memory and cache are not explicitly modeled, their performance effect is captured via the queueing delays at the CPU. The number of circulating jobs (i.e., the queueing network's population accordingly to the conventional

[4]We also did an experiment with 4 JVMs that confirmed that the right mix should be 2:2 and that resulted in resource utilizations as low as 75% for the equiutilization point.

[5]Response time is defined as the job end-to-end time, i.e., the elapsed time from its submission to its completion that includes all wait times at various resource queues.



(a) Two-station closed queueing model

(b) Three service demands

Figure 3. Schematics of system model and service demands.

queueing terminology [16]) corresponds to the number of JVMs in the system.

In [21], a closed form formula gives the application mix that optimizes the system power Φ, using as input parameters the service demands of each application per resource. These input parameters correspond to the aggregate time that each application uses each resource when executed in isolation, i.e., it does not reflect any waiting nor queueing times that are due to other applications. The model output provides the ratio $\boldsymbol{\beta} = (\beta_1, \beta_2)$, where β_1 corresponds to the percentage of class 1 applications and β_2 is the percentage of class 2 applications for any number of JVMs that execute simultaneously.

Applying this consolidation methodology on our two-station queueing model, we need to first obtain the input parameters of application j ($j = 1, 2$): its total CPU and disk service demands, $S_c(j)$ and $S_d(j)$, respectively. The optimal mix ratio $\boldsymbol{\beta^*} = (\beta_1^*, \beta_2^*)$ is then given by

$$\beta_1^* = \frac{\log \frac{S_d(2)}{S_c(2)}}{\log \frac{S_c(1)S_d(2)}{S_c(2)S_d(1)}}. \quad (2)$$

This is an optimal operational point, at which all resources are equally utilized. Moreover, when the system operates at such a point, both applications achieve their optimal performance, i.e., Φ, irrespective of the number of JVMs that are executing concurrently. If n is the number of JVMs, then the optimal population mix is $(n_1^*, n_2^*) = n \cdot (\beta_1^*, \beta_2^*)$. For more details on mathematics to obtain the optimal mix, we direct the interested readers to [21].

IV. Application Profiling : Service Demand Estimation

Critical for the model of optimal application mix is the calculation of application demands per resource, i.e., the total time an application spends on each of the resources excluding any waiting time. Such information is not straightforward to be collected. While there is a large body of work that focuses on sophisticated application profiling tools [8], it is unclear how to aggregate profiling data provided by low level performance counters (see Table I) into single values that represent the total time spent on specific resources. Here, we assume that an application consists of two basic phases: computation and IO. Service demands from a computational phase include the time to access the memory subsystem. Essentially we aggregate memory and cache access times with CPU times. CPU may be consumed during an IO phase also when the operating system handles read and write operations to disk. Based on the above discussion, we define three service demands for application j, which correspond to the accessed resources and application execution phases as follows:

- $S_c^c(j)$: CPU service demand, i.e., aggregate CPU time during a computational phase;
- $S_c^i(j)$: CPU service demand during IO, i.e., aggregate CPU time during an IO phase;
- $S_d^i(j)$: disk service demand, i.e., aggregate disk time during an IO phase.

Using the above terminology, the subscript denotes the system resource and the superscript corresponds to the application phase. In the rest of paper, we omit the index j when referring to a non-specific application. Figure 3(b) illustrates a possible relationship of phases of an application and corresponding accessed resources.

Note that S_c^i and S_d^i can be either completely or partially overlapped, depending on the hardware parallelism. Characterizing and estimating S_c^i and S_d^i is not trivial, especially within a virtualized environment [7], [10]. Our black box profiling manages to calculate these values from low level counters *without* instrumenting nor modifying the application.

A. Black Box Profiler

We estimate the CPU related service demands S_c^c and S_c^i by directly monitoring CPU utilization. Disk service demand is estimated based on the disk throughput. Our empirical studies show that the utilization of disk is not highly correlated with its throughput. Due to internal disk scheduling optimizations, disk utilization may be high but disk throughput may still be low and vice versa. This is a clear case where the utilization law [16] does not apply. Moreover, predicting IO demands only based on disk utilization is shown not to be optimal [25].

To obtain S_d^i, we require the collection of the following measurements during the execution of the application: (1) T, the total execution time of the application, (2) λ, the average rate of the disk read/write operations per second, and (3) t, the average time per disk read/write operation. Due to the concurrency available at the disk level, we estimate S_d^i as the the total application disk time normalized by $\gamma + 1$, the total number of requests at the disk, i.e., the maximum number of operations that the disk performs in parallel,

$$S_d^i = \frac{T \cdot \lambda \cdot t}{\gamma + 1}. \quad (3)$$

To estimate S_c^c, we just subtract S_d^i from T, because

computation and IO phases are sequential,

$$S_c^c = T - S_d^i. \quad (4)$$

The CPU service demand during the IO phase can be deduced by the difference between the total CPU busy time, i.e., the product of the total execution time T and the average core utilization, U_c, $T \cdot U_c$, and the CPU service demand during a computational phase, S_c^c,

$$S_c^i = T \cdot U_c - S_c^c. \quad (5)$$

To obtain the required statistics and estimate the service demands, we execute the application on the target system in isolation.

B. Service Demands on Two-station Queueing Model

Our aim is to collect data for a single instance of application that runs in isolation in the system, estimate its service demands S_c^c, S_c^i, and S_d^i and use this data as input in the model presented in Section III to predict the optimal application mix in the system. Accurate prediction of application end-to-end time when multiple instances of the same application are executing is the first step in validating the model correctness. Typically, in such a system, one could use mean value analysis (MVA) [16] to solve the simple queueing system shown in Figure 3(a) and predict application (i.e., end-to-end times) and system (i.e., utilization and throughput) performance. MVA has been extensively used for predicting application scalability [24], [29] but it does not directly apply to systems that are not subject to product-form solutions, e.g., when the service processes are bursty [18] or when the phenomenon of simultaneous resource possession exists [4]. Several efforts have focused on providing solutions to this problem by developing analytical models [4], [9], [23] that are based on combining MVA and Markov chain modeling, which requires a higher number of input parameters and introduces high computational complexity.

The specific issue we need to address here is simultaneous resource possession, as clearly shown in Figure 3(b). Instead of seeking an exact solution, we provide upper and lower bounds of application response times. Depending on the degrees of parallelism in the application program and system architecture, $S_d^i(j)$ and $S_c^i(j)$ can be overlapped or be sequential. For a given number of JVMs, the completely overlapped case gives the lower bound of the application response time, while the sequential case gives the upper bound.

1) Single Core System: To apply MVA on the two-station queueing system, the total CPU and disk service demands of an application are required. We use the total disk service demand by $S_d = S_d^i$, as the disk is only used during IO phase. The total CPU service demand depends on the degree of parallelism during the IO phase. If no parallelism exists, then one can compute the upper bound of the total CPU service demand by $S_{c\_up} = S_c^i + S_c^c$, i.e., the sum of CPU service demands during the computation and IO phases. On the other hand, when there is perfect parallelism in IO operations, one can compute the lower bound of total CPU service demand by $S_{c\_low} = S_c^c$.

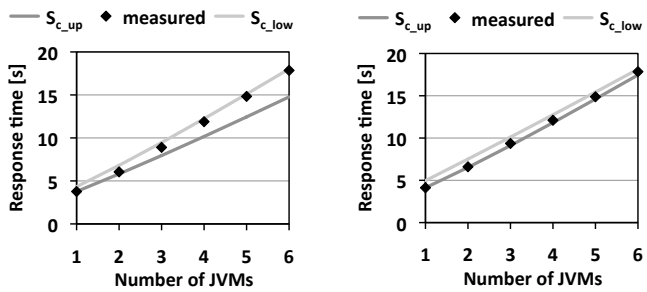


Figure 4. Upper and lower bounds of response times when executing multiple instances of microbenchmark A (on the left) and C (on the right).

In Figure 4, we depict the comparisons of measured response time, analytical upper bound with S_{c_up}, and analytical lower bound with S_{c_low} – both are analytically solved using MVA. Two types of applications, i.e., an IO bound and a CPU bound corresponding to microbenchmarks A and B in Section V, are presented. For the CPU bound application, both theoretical upper and lower bounds are very close to the measured data; whereas the difference is more prominent in IO bound case. This is due to the higher S_c^i values in IO bound applications. The measured data is closer to the lower bound for $n = 1$, where n is the total number of JVMs in the system and then gradually shifts to the upper bound. This experimental data leads us to use a step function for approximating the load dependent total CPU service demand as following,

$$S_c(j) \quad = \quad S_c^c(j) + \alpha S_c^i(j), \quad (6)$$

where $\alpha = 1$ for $n > 1$ and 0 otherwise.

2) Multicore System: The difference between single core and multicore systems relates to CPU service demand, S_c, as the number of disk remains the same in both cases. To obtain S_c^i and S_c^c of application j on a k-core system, we execute k instances of JVMs. To compute S_c, we further provide a scaling parameter to factor in the parallelism provided by the increasing number of cores. Theoretically, a k-core system can maximally scale down the CPU service demands by a factor of k. In practice, such a scaling factor depends (1) on the number of cores a single application's JVM can use simultaneously and (2) on the total number of JVMs in the system. Usually, a JVM can use as many cores as the number of concurrent threads in the application. As the benchmarks considered here are single threaded, a single JVM can only use up to one core. On the other hand, when the total number of JVMs, n, in the system is greater than k, a scaling factor

of k can be achieved. We propose to use the total CPU service demand being load-dependent and approximated by

$$S_c(j) = \frac{S_c^c(j) + \alpha S_c^i(j)}{\min\{n, k\}}, \quad (7)$$

where $\alpha = 1$ for $n > 1$, and 0 otherwise. One can easily see from Eq. 7 that the total CPU service demand is constant when $n \geq k$. As in the single core system, disk service demand is $S_d = S_d^i$. Note that the denominator in Eq. 7 explicitly factors in the interaction of cores and the number of instances.

V. Evaluation

In this section, we present the evaluation results of consolidation methodologies on an IBM Power 7 machine. All experiments are done using a single core and a quad-core system, to show that we effectively capture the parallelism offered by multiple cores. As our system has a single disk, we use up to four cores to avoid over saturating the disk while executing intensive workloads. To emulate a single and quad-core system out of eight cores, we use the `cpu_deallocate` command to restrict the number of available CPU cores. The evaluation methodology focuses on showing that the profiling approach is correct, that it effectively captures application scalability, and that the proposed optimal mix is indeed the correct one. The evaluation methodology is summarized as follows:

- We construct Java microbenchmarks where we can control the degree of their CPU/IO intensity. We use these benchmarks to validate the proposed black-box profiler. Then, we use the parameterization from profiling as input to the model and show that it predicts accurately the optimal mix.
- We continue by using applications from the DaCapo suite to show that our model captures their scalability well but also suggests the correct optimal mix, i.e., the consolidation strategy achieves the equal utilization point among resources and maximizes system power Φ.

A. Microbenchmarks

We design custom Java microbenchmarks that iterate through a computational phase, during which complex mathematical operations are executed, and an IO phase, during which the computed results are written to disk. We define the benchmark response time by the average time to complete one iteration and throughput as the average number of iterations completed per minute. The microbenchmark can be configured to have either CPU or IO bound behavior by specifying the duration of each phase.

1) Service Demand Estimation: Validation: We use benchmarks A and B on a single core system and benchmarks C and D on a quad-core system. We apply the black box profiler methodology via executing a single application on the single core system and four applications on the multicore. As such, for every application $j \in \{A, B, C, D\}$, we collect the service demands $S_c^c(j)$, $S_c^i(j)$, and $S_d^i(j)$. To validate our black box approach, we use code instrumentation, by inserting code snippets that collect time stamps at the beginning of each computational and IO phase to compute CPU and disk service demands. Naturally, such instrumentation is intrusive and difficult to apply in complex applications but we use it here to validate the accuracy of the black box service demand estimation.

micro-bench.	Response time [s]	Code instrumentation S_c^c	S_c^i	S_d^i	Black box S_c^c	S_c^i	S_d^i	cores
A	3.77	2.38	0.51	1.34	2.43	0.56	1.34	1
B	4.12	1.16	1.02	2.93	1.22	0.84	2.90	1
C	8.85	7.48	0.51	1.35	7.49	0.51	1.36	4
D	7.62	4.67	1.02	2.94	4.68	1.00	2.94	4

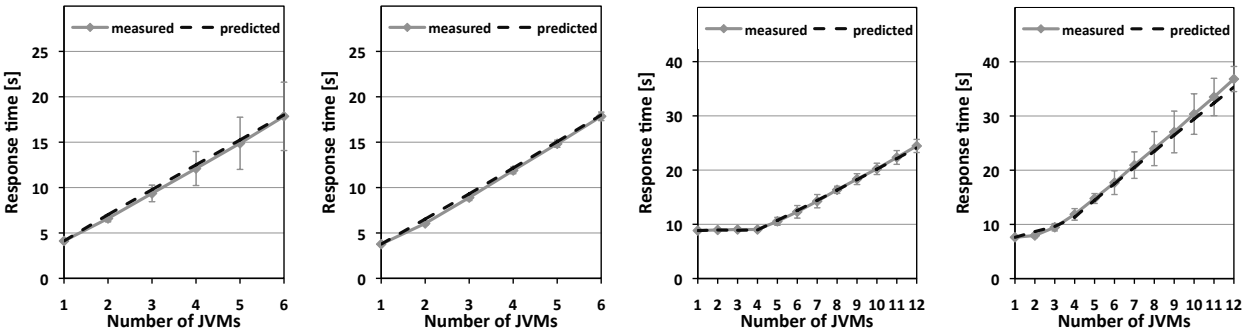
Table II
Profiling of microbenchmarks – CPU service demands and IO service demands via code instrumentation and via the black box approach.

We summarize the estimated CPU service demands and disk service demand in Table II, as well as the average response time, RT. Note that the RT obtained by code instrumentation is not necessarily exactly equal to $S_c^c + S_d^i$. In general, the black box approach results in similar values for S_c^c and S_c^i with code instrumentation. The difference in estimated disk service demand S_d^i between the two approaches is less than 1.5%.

We roughly classify the benchmarks into IO bound or CPU bound, depending on the ratio of S_d^i over RT, and the number of available cores[6]. We consider a benchmark being IO bound if $S_d/S_c > 1$, according to Eq. 6 and Eq. 7 for single core and quad-core, respectively. Therefore, benchmarks B and D are considered to be IO bound, and benchmarks A and C to be CPU bound.

2) Scalability Prediction of Microbenchmarks: We apply the proposed model of Figure 3(a) to predict the performance of a single type of applications with respect to different numbers of JVMs. We use values provided by the black box profiler, to compute the required input parameters in the queueing network and solve the system using the JMT tool [2]. JMT uses mean value analysis (MVA) [16] to predict application scalability on the system when multiple instances of the same application/benchmarks are executing simultaneously. Results are first validated with empirical values collected first on a single core system, then on the quad-core system.

[6]We have run a very extensive set of microbenchmarks where we vary the CPU/IO intensities so that we cover cases that are classified as very CPU intensive, to balanced in their CPU/IO demands, to very IO intensive but results are not presented here in the interest of space. We stress that the presented results are indicative for all microbenchmarks that we have used, i.e., the model is consistently accurate.



(a) RT(A) (b) RT(B) (c) RT(C) (d) RT(D)

Figure 5. Measured versus predicted average response time.

Figures 5(a) and 5(b) depict the predicted versus collected response time for application A and B, when the number of JMVs executing simultaneously ranges from 1 to 6. The error bars on the measured response time correspond to the standard deviation over 50 iterations. Overall, the model predicts the average benchmark response time very well.

Figures 5(c) and 5(d) depict the response time of microbenchmarks C and D, with the number of JMVs increasing from one to twelve. Similar to observation for the single core system, the model is in excellent agreement with experimental results. Note that the response time between 1 to 4 JVMs is roughly constant for microbenchmark C, because this benchmark is single threaded and each JVM can use up to one core. When the number of JVMs is higher than four, delays due to multiplexing are more visible and response time increases with the number of JVMs. For microbenchmark D, the response time slightly increases between one to four JVMs because of queueing at the disk (recall that there is a single disk in our experiments).

Summarizing, we have shown that the proposed black box approach is successful in estimating application resource demands at the various resources. In addition, we have shown that the model is successful in predicting accurately the response times of simultaneously executed benchmarks.

3) Optimal Consolidation of Multiple Types of Microbenchmarks: The real question that we are trying to answer here is how to consolidate different types of applications on the same system to maximize performance metrics. Here, we show that the consolidation strategy following Eq. 2 computes indeed the ideal mix between two classes of microbenchmarks and achieves equal utilization of CPU and disk. Due to lack of space, we do not present here analytical performance prediction results for job response times, system throughput, power, and utilization but we remark that they are consistently close to experimental numbers (i.e., errors are less than 5%).

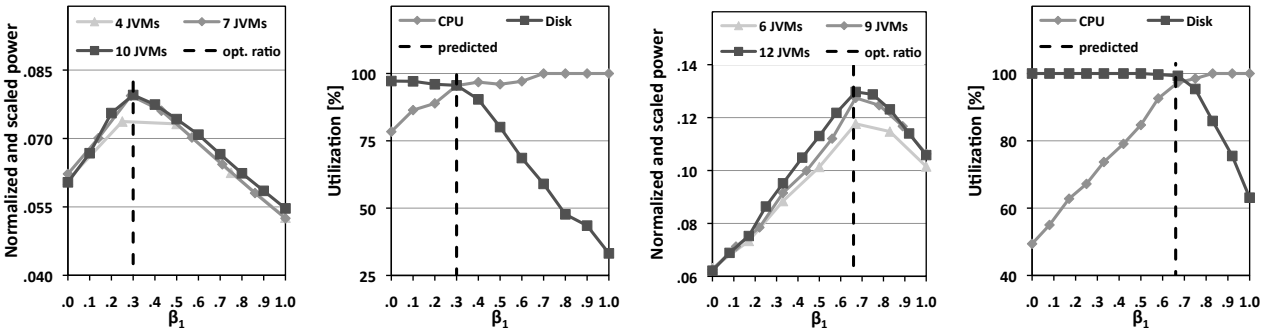
We use the service demands listed in Table II as given by the black box profiler to compute the optimal mix ratio (β_1^*, β_2^*) using Eq. 2. To validate the optimality of the mix, we empirically evaluate the normalized and scaled power Φ' [21] under different values of β_1 and for various numbers of JMVs. The normalized and scaled power Φ' is essentially $n \cdot \Phi$, defined in Eq. 1.

Figures 6(a) and 6(c) present the normalized and scaled power when consolidating microbenchmarks A with B, and C with D as a function of the consolidation parameter β_1. Results are presented for $n = (4, 7, 10)$ when we consolidating benchmarks A and B, and for $n = (6, 8, 10)$ for benchmarks C and D. The figure also presents utilizations of CPU and IO for $n = 10$ for both mixes, see Figures 6(b) and 6(d). The figure clearly shows that the model (see dashed line) consistently predicts accurately the mix that optimizes power as well as the system equiutilization point. This equiutilization point across all system resources guarantees that no single resource is the bottleneck.

B. DaCapo Benchmarks on a Quad-Core System

Using the DaCapo benchmarks listed in Table I, we show that our proposed methodology effectively predicts the response time of collocated Java applications and optimal operational point on a quad-core system. The three benchmarks are: 1) *batik*: a computation intensive program producing a number of Scalable Vector Graphics images based on the unit tests in Apache Batik, 2) *fop*: a computational intensive program parsing and transforming XSL-FO files to PDF files, 3) *luindex*: a relatively IO intensive program using *lucene*, an open source information retrieval software library, to index a set of documents, such as the works of Shakespeare and the King James Bible, 4) *h2*, a computational intensive program using a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, 5) *sunflow*, a computational intensive program rendering a set of images using ray tracing, and 6) *xalan*: a computational intensive program transforms XML documents into HTML.

1) Prediction of Scalability: Response Times of DaCapo Benchmarks : The black box profiler first executes 4 instances of JVMs of each benchmark on the quad-core system. Statistics are collected over an observation time of three minutes, excluding the first two minutes



(a) power(A, B) (b) util(A, B) (c) power(C, D) (d) util(C, D)

Figure 6. Consolidating different benchmark types on single core system (microbenchmarks A and B) and quad-core system (microbenchmarks C and D).

of warm-up time. The measured CPU and disk service demand metrics (S_c^c, S_c^i, S_d^i), are equal to $(1.87, 0.41, 0.11)$, $(0.50, 0.22, 0.07)$, $(1.00, 0.08, 0.71)$, $(,,)$, $(,,)$, and $(,,)$ for *batik*, *fop*, *luindex*, *h2*, *sunflow*, and *xalan*, respectively. Note that they are collected by executing one job in the system in isolation. Clearly, *batik* and *fop* are CPU bound benchmarks.Profiling suggests that *luindex* is also CPU intensive on a single core system as the total CPU service demand is higher than its disk demand *luindex*. When executing *luindex* on a quad-core system, the CPU demand decreases according to Eq. 7, and *luindex* becomes IO bound.

Figure 7 shows the response times as a function of the number of simultaneously executing DaCapo benchmarks. The figure reports both experimental and predicted values. Similar to CPU intensive microbenchmarks, the response times of *batik* and *fop* are nearly constant for 1 to 4 JVMs and increase from 4 JVMs onward. Our predictions are quite accurate for 4 to 9 JVMs, slightly over-estimated for 1 to 3 JVMs, and slightly under-estimated for 9 JVMs onward. For *luindex*, our prediction is always slightly higher, especially for 1 to 4 JVMs. This is an outcome of disk runtime optimizations, which depend on the disk load. Overall, the model captures well the performance trends.

As *h2*,*sunflow* and *xalan* are longer benchmarks and the standard deviation of measurements is very small, one can not easily observe the confidence interval as other benchmarks. For *sunflow*, the prediction is fairly accurate, whereas the prediction of a larger number of VMs is visibleness lower than the measured one for *h2* and *xalan*.

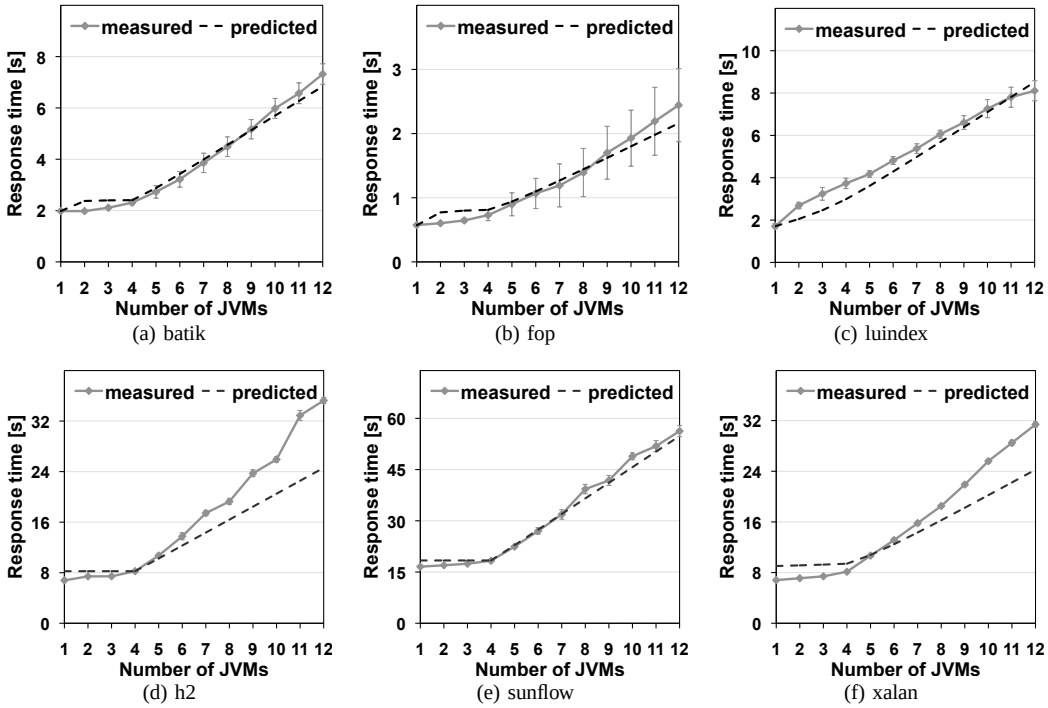
2) Optimal Consolidation: We consider two consolidation mixes: (1) *batik* with *luindex* and (2) *fop* with *luindex*. Our methodology predicts the optimal mix ratio β_1 to be 0.4 and 0.5 for *batik* and *fop*. Figure 8 depicts the normalized and scaled power with respect to different ratios of *batik* and *fop* and different number of total JVMs. As shown also in the motivating example, see Figure **??** in Section II, the experimental values for CPU and disk utilizations show that equiutilization is achieved for the *fop* and *lunindex* mix when $\beta_1 = 0.5$. Similarly, when consolidating *batik* and *luindex*, the experimental equiutilization point is for $\beta_1 = 0.4$, As noted earlier, our methodology also produces analytical response times, power, and utilization curves for the consolidated DaCapo workloads. These analytical results are in very good agreement with experimental values and are not shown here in the interest of space.

Summarizing, the experiments with DaCapo benchmarks strongly argue for the model's robustness. The results presented in this section for the optimal consolidation of DaCapo benchmarks are representative. We have done exhaustive experiments on all possible mixes from the various DaCapo benchmarks and our results confirm that the modeling methodology accurately predicts the optimal mixing point, strongly arguing for the robustness of the proposed methodology.

VI. Related Studies

There is a large body of work developing applications and workload management [15], [17], [22], [25] to consolidate applications in data centers, at the level of operating System Virtual Machine (VM). Two common foci are: (1) profiling and predicting VM resource demands and (2) developing heuristic based VMs allocation algorithms. Other related work focuses on designing real time scheduling algorithms for VMs [7], [10], [20], with the emphasis on improving IO operations. Note that commonly used vertical profiling tools [8] in the software engineering domain can identify a set of performance-relevant parameters, but there is a lack of systematic approaches to further predict performance interferences of consolidated applications.

Wood and Shenoy developed Sandpiper, which implements two profiling approaches (1) a black box approach, i.e., fully OS- and application-agnostic and (2) a gray-box approach exploiting OS- and application-level statistics. Wood et. all [25] used a regression-based model to profile and predict application resource requirements in a visualized environment. Their regression model can map the native system profile into a virtualized one and predict the VM CPU utilization. In this work, the authors also showed that



(a) batik (b) fop (c) luindex

(d) h2 (e) sunflow (f) xalan

Figure 7. Predicted and measured average response times of DaCapo applications on quad-core system.

the complexity of modeling IO operations is challenging. Lu et. al. [15] developed a profiling methodology that views the problem of physical resource utilization as a the source separation problem in digital signal processing, and designs a directed factor graph (DFG) to successfully model the dependence relationships among different resources (CPU, memory, disk, network) across virtual and physical layers.

Gmach et. al [5] used a trace based approach for capacity management and demand prediction for single application's OS-VMs data centers. Kraft et. al. [13] also use a trace-driven approach to predict scheduling delays of IO requests due to consolidation in virtualized environments. Meng et. al. [17] proposed a joint OS-VM provisioning approach in which multiple VMs are consolidated and provisioned together, based on an estimate of their aggregate capacity needs. The capacity of joint OS-VM is defined through the service level agreement and estimated via a combination of prediction techniques, linear regression, ARMA, and neural networks. Govindan et. al. [7] developed a CPU scheduling algorithm for virtual machine monitoring. Their scheduler incorporates the IO behavior of OS-VMs such that the network activities of multi-tier applications can be fairly scheduled.

Performance prediction of Java based applications are developed with great detail for software components [30]. Abstract queueing models have been applied on predicting performance of specific software components [1], [14], [30]. Prediction models often use Mean Value Analysis (MVA), a simple and efficient technique that proved very useful for performance prediction of modern computing systems such as three tiered web servers [29]. MVA can be applied to obtain performance measures of single and multiple applications. In general, little is known on optimizing system performance by mixing multiple classes of applications/workloads. To the best of our knowledge, the only theoretical methodology that focuses on this problem is the one presented in [21].

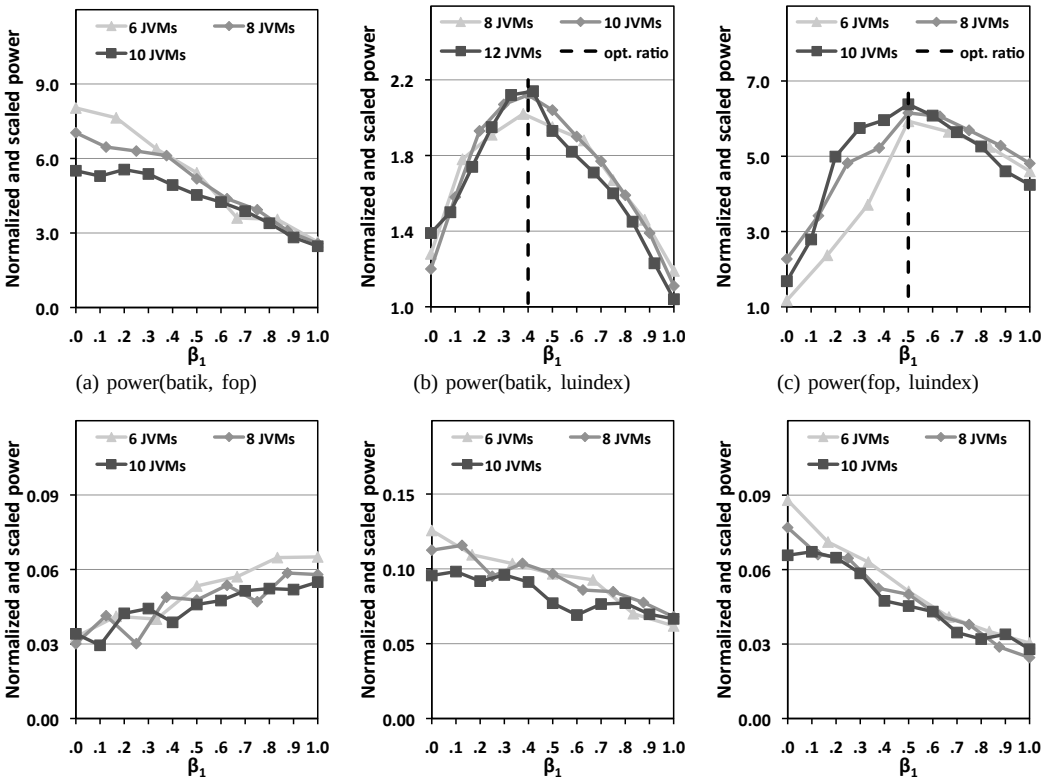
In this paper, we depart from prior work by applying the theoretical methodology in [21] to solve the difficult problem of predicting performance in a multi-core system where different classes of Java applications are consolidated. Our profiling methodology is lightweight and at a higher level than the OS-VM of the system stack and is critical for correct model parameterization. The presented model-driven consolidating strategy is complimentary to existing VMs consolidation and scheduling algorithms. Our experimental results on the reference system show accurate prediction of Java application performances in the order of seconds, whereas most of performance models of VMs are validated only analytically or via trace-driven simulation.

VII. Conclusion

Maximizing resource utilization on modern multicore systems hinges on the scalability of executing applications.

Efficient profiling of applications and accurate performance prediction are prerequisite to finding the system's

optimal operational point, i.e., maximizing throughput while minimizing



(a) power(batik, fop) (b) power(batik, luindex) (c) power(fop, luindex)

(d) power(h2, sunflow) (e) power(h2, xalan) (f) power(sunflow, xalan)

Figure 8. Normalized and scaled power for two DaCapo mixes for different values of simultaneously executing JVMs on a quad-core system.

response times for any mix of applications.

In this paper, we first provide a simple black box profiler to quantify the service demands of CPU- or IO-bound applications. We present a closed queueing network to approximate and predict the performance of multiple applications for multicore systems. Our evaluation results show that we can accurately predict application performance of different application mixes on our reference system.

Concerning ongoing research, we are extending our methodology to better model disk optimizations, to deal with multi-threaded applications in a dynamically changing environment, as well as with different performance targets, such as Service Level Objectives (SLOs).

Preliminary results with experimentation on all DaCapo benchmarks further confirm that the proposed modeling methodology is practical and promising

when considering a variety of performance targets.

REFERENCES

[1] S. Balsamoa, A. Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Trans. Softw. Eng.*, 30:295–310, 2004.

[2] M. Bertoli, G. Casale, and G. Serazzi. JMT: Performance Engineering Tools for System Modeling. *SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.

[3] S. Blackburn, R. Garner, C. Hoffman, A. Khan, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, , D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of OOPSLA*.

[4] D. Freund and J. Bexfield. A New Aggregation Approximation Procedure for Solving Closed Queueing Networks with Simultaneous Resource Possession. In *Proceedings of ACM SIGMETRICS*, pages 214–223, 1983.

[5] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Capacity Management and Demand Prediction for Next Generation Data Centers. In *Proceedings of International Conference on Web Services*, ICWS, pages 43–50, 2007.

[6] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. Technical report, Microsoft, 2011.

[7] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: Communication-aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *Proceedings of International Conference on Virtual Execution Environments*, VEE, pages 126–136, 2007.

[8] M. Hauswirth, P. Sweeney, A. Diwan, and M. Hind. Vertical Profiling: Understanding the Behavior of Object-oriented Applications. In *Proceedings of OOPSLA*.

[9] P. Heidelberger and K. S. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Trans. Comput.*, 31:1099–1109, 1982.

[10] H. Hu, X. Long, J. Zhang, J. He, and L. Xia. I/O Scheduling Model of Virtual Machine Based on Multi-core Dynamic Partitioning. In *Proceedings of the Symposium on High Performance Distributed Computing*, HPDC, pages 142–154, 2010.

[11] D. Kaseridis, J. Stuecheli, J. Chen, and L. K. John. A Bandwidth-Aware Memory-subsystem Resource Management Using Non-invasive Resource Profilers for Large CMP Systems. In *HPCA*, pages 1–11, 2010.

[12] Y. Koh, R. C. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of ISPASS*, pages 200–209, 2007.

[13] S. Kraft, G. Casale, D.Krishnamurthy, D. Greer, and P. Kilpatrick. Performance Prediction in Consolidated Virtualized Environments. In *Proceedings of International Conference on Performance Engineering*, ICPE, pages 295–306, 2011.

[14] T. Liu, S. Kumaran, and Z. Luo. Layered Queueing Models for Enterprise JavaBean Applications. In *Proceedings of International Conference on Enterprise Distributed Object Computing*, EDOC, pages 174–178, 2001.

[15] L. Lu, H. Zhang, G. Jiang, H. Chen, K. Yoshihira, and E. Smirni. Untangling Mixed Information to Calibrate Resource Utilization in Virtual Machines. In *Proceedings of International Conference on Autonomic Computing*, ICAC, 2011.

[16] D. Menascé, V. Almeida, and L. Dowdy. *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Prentice Hall, 1994.

[17] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis. Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. In *Proceedings of International Conference on Autonomic Computing*, ICAC, pages 11–20, 2010.

[18] N. Mi, G. Casale, L. Cherkasova, and E. Smirni. Burstiness in Multi-tier Applications: Symptoms, Causes, and New Models. In *Middleware*, pages 265–286, 2008.

[19] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: Managing Performance Interference Effects for QoS-aware Clouds. In *Proceedings of European Conference on Computer systems*, EuroSys, pages 237–250, 2010.

[20] A. Povzner, D. Sawyer, and B. S. Horizon: Efficient Deadline-Driven Disk I/O Management for Distributed Storage systems. In *Proceedings of Symposium on High Performance Distributed Computing*, HPDC, pages 1–12, 2010.

[21] E. Rosti, F. Schiavoni, and S. Giuseppe. Queueing Network Models with Two Classes of Customers. In *Proceedings of Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '97, pages 229–234, 1997.

[22] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource Allocation Algorithms for Virtualized Service Hosting Platforms. *J. Parallel Distrib. Comput.*, 70:962–974, 2010.

[23] D. Towsley, K. Chandy, and J. Browne. Models for Parallel Processing within Programs: Application to CPU: I/O and I/O: I/O Overlap. *Commun. ACM*, 21:821–831, 1978.

[24] B. Urgaonkar, G. Pacifici, P. S. M. Spreitzer, and A. Tantawi. An Analytical Model for Multi-tier Internet Services and Its Applications. In *Proceedings of International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, pages 291–302, 2005.

[25] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proceedings of International Conference on Middleware*.

[26] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Sandpiper: Black-box and Gray-box Resource Management for Virtual Machines. *Comput. Netw.*, 53:2923–2938, 2009.

[27] Y. Xie and G. H. Loh. PIPP: Promotion/insertion Pseudo-partitioning of Multi-core Shared Caches. In *Proceedings of ISCA*, pages 174–183, 2009.

[28] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy. Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates. *SIGSOFT Softw. Eng. Notes*, 31, 2005.

[29] Q. Zhang, L. Cherkasova, N. Mi, and E. Smirni. A Regression-based Analytic Model for Capacity Planning of Multi-tier Applications. *Cluster Computing*, 11:197–211, 2008.

[30] R. Zhang, Z. Budimlic, and K. Kennedy. Performance Modeling and Sreditction for Scientific Java Applications. In *Proceedings of ISPASS*, 2006.