Achieving Application-Centric Performance Targets via Consolidation on Multicores: Myth or Reality?

Lydia Y. Chen
IBM Zurich Lab
Switzerland
yic@zurich.ibm.com

Danilo Ansaloni
University of Lugano
Switzerland
danilo.ansaloni@usi.ch

Evgenia Smirni
College of William and Mary
USA
esmirni@cs.wm.edu

Akira Yokokawa
University of Lugano
Switzerland
akira.yokokawa@usi.ch

Walter Binder
University of Lugano
Switzerland
walter.binder@usi.ch

ABSTRACT

Consolidation of multiple applications with diverse and changing resource requirements is common in multicore systems as hardware resources are abundant and opportunities for better system usage are plenty. Can we maximize resource usage in such a system while respecting individual application performance targets or is it an oxymoron to simultaneously meet such conflicting measures? In this work we provide a solution to the above difficult problem by constructing a queueing-theory based tool that we use to accurately predict application scalability on multicores and that can also provide the optimal consolidation suggestions to maximize system resource usage while meeting simultaneously application performance targets. The proposed methodology is light-weight and relies on capturing application resource demands using standard tools, via non-intrusive low-level measurements. We evaluate our approach on an IBM Power7 system using the DaCapo and SPECjvm benchmark suites where each benchmark exhibits different patterns of parallelism. From 900 different consolidations of application instances, our tool accurately predicts the average iteration time of collocated applications with an average error below 10%.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Modeling and Prediction*

Keywords

Performance modeling, consolidation, multicores.

1. INTRODUCTION

Multicore architectures hosting multi-threaded applications are the prevailing execution environments in today's data centers and cloud computing facilities. Modern multicore systems are characterized by large computational capacity, and deep memory hierarchies, and are well-equipped with hardware specific acceleration and optimization techniques. Various programming models are thus developed to optimize application execution on heterogeneous architectures. Often, a single application instance executing on such a powerful platform under-utilizes system resources [18, 23]. To increase resource efficiency, system providers resort to consolidation, i.e., packing multiple applications into a physical server, lowering the operating cost and augmenting system throughput [11, 20].

Typically, applications can be classified based on their concurrency level, intensity of resource demands, and performance level objectives. For example, certain applications may have stringent performance requirements especially if they are associated with higher service tiers [20], whereas some applications do not require any performance guarantees. In this paper, we consider two kinds of application consolidation: *homogeneous* consolidation, where multiple application instances of the same type are executed simultaneously and *heterogeneous* consolidation where different application instances are selected to execute simultaneously. Naturally, consolidation aims to avoid the perils of resource oversubscription under single instance execution and achieve a better utilization of resources [7]. This becomes a tough technical challenge, especially given the need of differentiated performance objectives per application.

In this paper, we aim to meet pre-set per application requirements in the form of average execution times while the instances of simultaneously executing applications are maximized. The challenge is to accurately *predict* performance interference among collocated applications on multicores [20, 15, 9], and based on this prediction to reach judicious decisions regarding this difficult problem. A posteriori, one could decide on the best consolidation if an exhaustive experimental search of all possible combinations is made. This may be a viable solution if only a small set of applications is executing on the system. Here, we advocate using a modeling-based methodology that can shed light on achieving a priori optimal consolidation, especially in terms of efficient and accurate predictions of performance objectives such as average application iteration time and system throughput. Central to the model's effectiveness is the ability to encapsulate the workload dynamics, executed over complex hardware and software stacks.

There are several related works [6, 22] that focus on models to capture low level end-to-end performance metrics, such as absolute or relative Instructions Per Cycle (IPC). Low-level models can provide a detailed overview of the resource demands of a single application execution at the cost of exploring a large number of hardware counters. Extracting the essential characteristics for application performance from state-of-the-art machines is a complex engineering task because of the multiple problem dimensions, including the execution of application threads on different cores, accesses to shared data structures, inter/intra thread communication, and runtime optimizations. In addition, low-level performance measures usually do not directly reflect application iteration time[1] among consolidated instances, neither can they convey information about the expected average iteration time.

In this paper, our objective is to identify optimal homogeneous or heterogeneous consolidations by accurately suggesting the maximum number of application instances to consolidate without violating target iteration times. We develop a model-driven approach that predicts iteration time under various consolidations and validate our results using the widely used DaCapo benchmark suite [5] on an IBM Power7 system. We first develop a light-weight and non-intrusive profiling methodology to compute application resource demands. The profiling methodology effectively captures the application concurrency level, the hardware parallelism, and the impact of resource runtime optimizations. We show that resource demands may vary according to the number of consolidated instances and that these variations are captured by the model. Performance interference among consolidated instances is captured via a surprisingly simple two station queueing model that is appropriately parameterized, thanks to the developed profiling methodology. We extend the mean value analysis algorithm for multi-class systems [17] to allow for load-dependent service demands and demonstrate the accuracy of the new model.

To summarize, our contributions are both theoretical and practical. On the theoretical side, we present an extension to the Mean-Value-Analysis (MVA) algorithm allowing for analytic computation in load dependent multi-class queueing networks that could only be solved via simulation. On the practical side, we present a light-weight profiling methodology that extracts vital resource demand information using standard profiling tools and use this information to parameterize the model. This model effectively captures performance interference, accurately predicts program execution time, and suggests ideal consolidations. From more than 500 experiments of homogeneous and heterogeneous consolidations of DaCapo benchmarks, the model achieves average prediction errors below 8%, further illustrating the model's robustness. We note that as our methodology is generic and platform independent, it can readily apply on different workloads and/or different platforms.

[1]Because the applications that we use to evaluate the methodology proposed in this paper are composed by a set of iterations, we use the average "iteration time" as a measure of the application end-to-end execution time. Effectively, a scaled iteration time (multiplied by the number of iterations) expresses the application execution time.

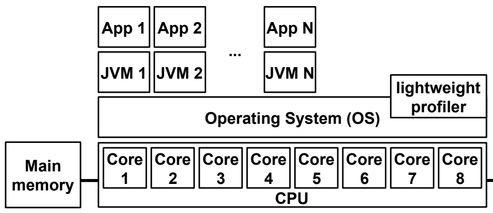


Figure 1: High-level view of the execution environment.

This paper is organized as follows. Section 2 gives an overview of homogeneous and heterogeneous consolidation. The proposed profiling methodology for capturing resource demands is described in Section 3. The queueing model and the extension of the mean value analysis algorithm is given in Section 4. Section 5 presents experimental results. Related work is given in Section 7, followed by the conclusion in Section 8.

2. CONSOLIDATION

The performance of an application depends not only on how application threads use the underlying hardware resources but also on the interference of collocated applications. In this section, we illustrate the complexity of consolidation using the DaCapo benchmark suite [5] on our target hardware platform, an IBM POWER7 server.

2.1 Reference System and Workload

The reference system is an IBM Power 750 Express server with a single POWER7 processor board hosting 8 cores running at 3.00 GHz, SMT set to 1, 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 a 64 MB buffer. The system runs a logical partition with AIX 6.1 (64 bit) and IBM J9 JVM SR8-FP1 (64 bit) in server mode, with 2 GB heap size and default garbage collection algorithm. Figure 1 provides a high-level view of the system, omitting network components as we limit our observations to non-network intensive applications.

We first developed our methodology using a set of micro-benchmarks to test our predictions. This initial set of reference programs[2] included both Java and native workloads, giving us fined-grained control over the amount and the duration of CPU and disk operations. Due to the space limit, we skip the presentation of micro benchmark results.

In this paper, we mainly evaluate our methodology with workloads from the widely-used DaCapo 9.12 benchmark suite [5] and demonstrate the complexity of the workload dynamics. The Dacapo suite is representative of contemporary Java workloads and consists of fourteen benchmarks with various levels of parallelism. Here we focus on a selection of 10 benchmarks, excluding network intensive benchmarks (i.e., `tomcat`, `tradebeans`, and `tradesoap`) and benchmarks with high iteration time (i.e., `eclipse`). Extending the proposed technique to network intensive applications is subject of future work.

All empirical measurements presented in this paper are based on a warm-up time of 2 minutes and an observation

[2]Throughout this paper we use the terms "program", "benchmark" and "application" interchangeably.

Table 1: Concurrency in DaCapo benchmarks

Benchmark	no. of threads				
	total	alive		runnable	
		peak	average	peak	average
`avrora`	11	11	10.90	8	4.87
`batik`	13	8	6.93	6	3.00
`fop`	6	6	4.93	5	3.01
`h2`	8	7	5.65	5	3.22
`jython`	6	6	4.98	5	3.00
`luindex`	6	6	5.07	5	3.01
`lusearch`	6	6	5.96	5	3.01
`pmd`	14	13	8.90	12	4.04
`sunflow`	10	8	6.99	5	3.00
`xalan`	6	6	5.97	6	3.01

time of 3 minutes. To this end, we configure the benchmark harness to execute an infinite amount of iterations in the same JVM process[3]. We do not collect any metrics during the warm-up phase, as performance is affected by class-loading and just-in-time compilation. The average iteration time is computed as the average amount of time between consecutive benchmark iterations started after the beginning of the observation time. Garbage collection is not explicitly triggered between the iterations, the just-in-time compiler is always turned on, and we do not bind any process or thread to specific cores.

Table 1 summarizes the internal parallelism of each benchmark: total number of threads (including JVM threads), peak and average number of alive threads (including idle threads), and peak and average number of runnable threads (including running threads). The presented values are computed over an entire benchmark iteration using the Java Management and Monitoring API and a custom native agent.

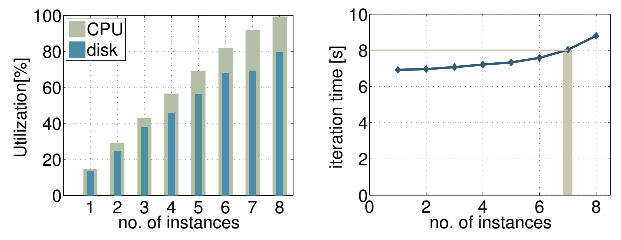
The DaCapo benchmarks depict wide variability in their concurrency levels. Some benchmarks have a higher (lower) degree of concurrency, e.g. `pmd` (`luindex`), whereas some others have a higher (lower) volume of thread communications, e.g., `avrora` (`lusearch`). Each benchmark differs in hardware resource usage, i.e., some are CPU bound and some are disk bound. Due to the complexity of hardware architecture and applications, it is challenging to identify the key characteristics that dictate the iteration times per application, and even more challenging when consolidating multiple instances of applications. In the following, we illustrate the difficulties of finding effective consolidation of two DaCapo benchmarks: `xalan` and `luindex`.

2.2 Problem Statement with Examples

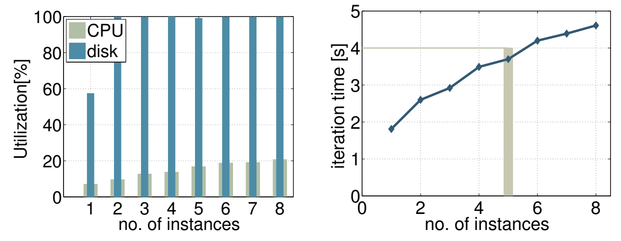
The problem addressed by this study is how to find the optimal (maximum) consolidation on multicore systems without violating target iteration times of individual applications. We consider two kinds of consolidations, homogeneous and heterogeneous, that raise several questions:

Homogeneous Consolidation: For a given application and hardware, what is the maximum number of instances one can simultaneously execute without violating the target iteration time? In particular, how does the concurrency pattern and underlying resource usage of the application affect this number? We illustrate this problem by consolidating multiple instances of `luindex` and `xalan`, see Figure 2(a) and 2(b).

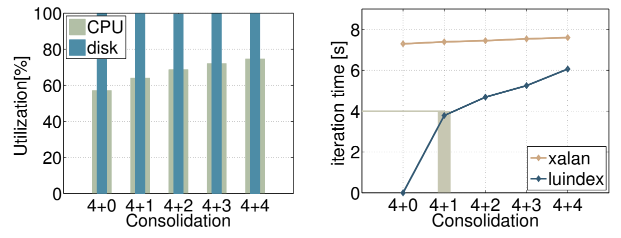
[3]Throughout this paper we use the terms "JVM process" and "application instance" interchangeably.



(a) consolidating `xalan`



(b) consolidating `luindex`



(c) consolidating `xalan` and `luindex`

Figure 2: Resource utilizations and iteration times under homogeneous and heterogeneous consolidation. For the heterogeneous consolidation (third row of graphs), we assume that there are 4 instances of `xalan` consolidated with 1, 2, 3, and 4 instances of `luindex`.

Heterogeneous Consolidation: We consider that a system executes two heterogeneous applications, a primary and a secondary application. Given a fixed (required) number of execution instances and target iteration time for the primary application, what is the maximum number of consolidated instances of the secondary application so that the target iteration time of the primary application is not violated? How can an optimal consolidation leverage the complementary resource usage and concurrency patterns of both applications? We illustrate this problem by consolidating four instances of `xalan` (primary) with different instances of `luindex` (secondary) ranging from one to five.

To find the optimal consolidations for the above two problems we conduct exhaustive experiments. Figure 2 shows how CPU, disk utilization and the iteration times behave for several consolidations of `xalan`, `luindex`, and `xalan+luindex`. Let the target iteration times be 4 and 8 seconds for `luindex` and `xalan`, respectively. To meet the target iteration time, one can find that the optimal consolidation instances for `luindex`, `xalan` and `xalan+luindex` are 5, 7, and 4+1, respectively. As `xalan` and `luindex` are CPU- and disk intensive applications, their optimal homogeneous consolidation results in high CPU utilizations for `xalan` and high disk utilizations for `luindex`. The heterogeneous con-

Table 2: **CPU related statistics under different consolidated instances of `batik` and `avrora`**

instances	batik			avrora		
	T [s]	U_c [%]	U_{cs} [%]	T [s]	U_c [%]	U_{cs} [%]
1	2.08	13.6	93	7.47	33.2	92
2	2.09	26.9	97	7.88	49.4	93
4	2.09	53.3	98	11.00	60.8	95
6	2.09	77.6	99	12.38	74.8	97
8	2.15	96.4	100	14.29	85.2	100
10	2.65	99.6	100	16.39	91.2	100
12	3.14	99.9	100	18.85	94.9	100
14	3.70	100.0	100	21.55	97.0	100
16	4.23	100.0	100	24.19	98.2	100

solidation results instead in more "balanced" CPU and disk utilization levels.

Clearly, optimal consolidation depends on how the workload increases with each additional instance, how resource run-time optimizations are affected by collocation, and how strong performance interference is among competing applications. The prediction methodology should suggest a priori and without exhaustive experimentation the ideal consolidation. Prerequisite to the above target is the development of a methodology that can encapsulate the critical application characteristics. This is the subject of the following section.

3. CHARACTERIZING THE WORKLOAD

The concept of "resource demand", i.e., the time an application spends on CPU and disk resource, has been widely used in characterizing applications on multicore systems [6, 22] as well as on multi-tier and virtualized environments [25, 19, 18]. An application iteration completes by executing multiple threads, which can concurrently or sequentially access the system resources. The iteration time consists of the sum of execution times on the *distinct* resources, which are defined as resource demands, and the wait times on these resources due to contention caused by collocation. The aim here is to characterize the resource demands per application under consolidation scenarios. Note that we also refer to resource demands as execution times *without any consolidation.*

Most of the related studies focus on obtaining resource demands of a single application instance using very fine grained and detailed information, such as thread communication [6] and cache related statistics [11]. To this end, the standard approach is to explore a large number of hardware counters or even modify the application source code. Such a process is cumbersome, intrusive, and neither portable nor scalable. Using hardware performance counters is clearly the preferable approach but it is challenging to collapse such diverse information into a single value parameter such as the resource demand. In the following two subsections we illustrate how exactly we achieve the above target via a lightweight profiling methodology that can capture resource demands as a function of collocated workloads on different resources.

3.1 CPU Demand

On a single core system, the CPU demand can be obtained by the average CPU utilization, defined as the fraction of time the single core is busy during the execution. If there are multiple instances executing simultaneously and the CPU is not fully saturated, then one may assume that the CPU is shared equally among the competing instances. If the CPU is saturated, waiting (queuing) time kicks in and contributes to the application iteration time. Therefore, CPU utilization is a good indicator of CPU demand per application instance, but not of the application iteration times.

For multicore systems, the CPU demand is not only defined by core utilization, but also by the time that *at least* one core is busy. To obtain the CPU demand of multithreaded applications on multicores, we propose combining two performance counters: CPU utilization, U_c, which denotes the aggregate busy time of all cores and (CPU) system utilization U_{cs}, which denotes the fraction of time at least one thread is executing. On UNIX-like systems, those values can be collected using the `mpstat`, `sar`, and `vmstat` commands. Note that although the memory is another important resource, we encapsulate the time accessing the memory [11] simply within the CPU times, i.e., these two resources are coalesced as one.

We illustrate the CPU profiling using two DaCapo benchmarks, `batik` and `avrora`. Table 2 lists U_c, U_{cs}, and their iteration times, denoted by T, for different consolidated instances (homogeneous consolidations), with the number of instances ranging from 1 to 16. Looking at results for a single instance, a high difference between the two utilization values suggests that only a small set of cores are busy at a given moment in time, due to limited concurrency level within each application. When all cores are busy processing multiple instances, CPU utilization U_c is the same as the CPU system utilization U_{cs}. The minimum number of instances that keep all cores busy is considered the system *saturation point* for this specific homogeneous consolidation. In summary, U_c, U_{cs} and the saturation point are critical for our workload characterization as well as model development.

3.1.1 Linear Increment of U_c

From Table 2, one can observe that CPU utilization increases almost linearly and indicates that the CPU demand per additional instance is constant. From the constant iteration times, one can further infer that the additional CPU demand per instance is distributed to the idle cores.

In the case of linear demand increment, one can compute the CPU demand with respect to collocated instances as follows: When executing one instance of `batik`, CPU demand is $0.93 \cdot T$. When two instances are executing, the CPU demand is $0.93 \cdot T/2$. One can do similar calculations up to 8 instances. After that point the CPU approaches saturation and iteration times start increasing. In the `batik` case, we see that the value of such a point is roughly equal to the inverse of U_c of a single instance execution. In the following subsection we look at another typical case, that of `avrora`, where U_c does not increase linearly with the number of instances. In the following section we examine the reasons for this behavior.

3.1.2 Nonlinear Increment of U_c

In contrast to `batik`, the CPU utilization U_c of `avrora` grows non-linearly and its iteration times increase as the number of instances increases, even under a non-saturated situation, see Table 2. This behavior, observed also in `pmd`, is initially perplexing as it seems that the CPU demand contributed by each instance *decreases*, perhaps due to some run-time optimization. Extrapolating the saturation point from a single instance as we did successfully in `batik` would result in a significant error here, as the predicted saturation

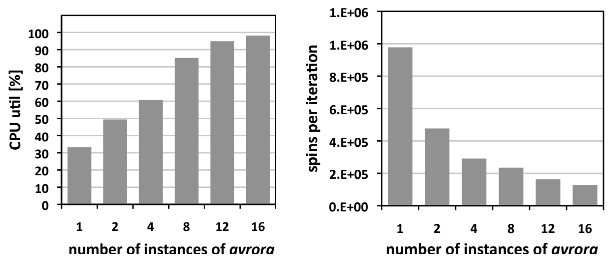


Figure 3: CPU utilization and number of spins depending on the number of instances of `avrora`.

point would be three instances, while in reality the saturation point is much larger. To obtain a good estimate of the saturation point, additional profiling is required.

`avrora` has a high number of threads, performing frequent, fine-grained interactions with each other. It intensively uses synchronization and spin locks, which are commonly used to protect specific code regions. Unlike standard locks, a thread that fails to acquire a spin lock does not get immediately descheduled, but it repeatedly tries to acquire the lock for some time, provided that computational resources are available. We conjecture that the CPU demand per instance can vary depending on the use of spin locks and idle cores.

Figure 3 presents the CPU utilization and number of spins with respect to different numbers of consolidated instances of `avrora`. The number of spins is collected via the Java Locking Monitor (JLM) tool. As the consolidated instances increase, the percentage of idle cores decreases, so do the spins per interaction. Such an observation confirms our conjecture that the CPU demand per instance decays due to a lower number of spins when fewer idle cores are available. Among the benchmarks considered in this study, `avrora` and `pmd` show non-linear increases in CPU utilization as consolidated instances increase due to the use of spin locks. We thus propose to monitor the number of spins and if this number is low, then we assume that we are in a "linear CPU demand increment" case as in `batik` and the saturation point is updated accordingly. If the number of spins is high, then we need more information to compute the saturation point. We address this issue in the following subsection.

3.1.3 Proposed Profiling for CPU Demand

Following analysis from previous subsection, we summarize the algorithmic description of CPU demand. Let $D_c(n)$ denote the CPU demand per iteration when n instances are consolidated. Let $U_c(n)$ and $U_{cs}(n)$ denote the CPU utilization and CPU system utilization respectively, given n consolidated instances. Moreover, we denote the saturation point as ξ, which represents the minimum number of instances that completely saturate the CPU. Depending on the number of spins, ξ can be decided as follows.

We estimate the per instance CPU demand under n consolidated instances, $D_c(n)$, by the product of iteration time $T(1)$ and CPU utilization $U_{cs}(1)$ under a single instance execution, divided by the minimum of the saturation point, ξ, and the number of instances, n:

$$D_c(n) \quad = \quad \frac{T \cdot U_{cs}(1)}{\min(n, \xi)} = \frac{D_c(1)}{\min(n, \xi)}. \tag{1}$$

Both $T(1)$ and $U_{cs}(1)$ are measured via profiling the single instance execution. One can also view the denominator of Eq. 1 ($\min(n, \xi)$) as the maximum embedded parallelism.

To calculate the saturation point ξ, we first monitor the number of spins in a single instance execution. The saturation point $\xi(1)$, is estimated by the inverse of CPU utilization, i.e., $\frac{1}{U_c(1)}$. When the number of spins is below a threshold, we let $\xi = \xi(1)$; otherwise we re-estimate the new saturation point for non-linear demand increment, by executing $\lceil \xi(1) \rceil$ application instances. Note that the threshold for monitoring spins is obtained via prior empirical experiences. We then recompute the average CPU utilization per instance and take the inverse of such a value as the new estimate of the saturation point.

3.2 Profiling Disk Demand

Measuring the performance of the disk is particularly difficult due to the multiple buffers present at all storage levels, the high number of run time optimizations, e.g., out-of-order writes, and parallel writing across different disk platters. Profiling of disk demand must incorporate the inherent disk parallelism due to consolidation.

Here, we rely on disk operation statistics, instead of disk utilization, because the latter is a biased metric due to multiple I/O buffering [26]. To compute the disk demand under a single application instance $D_d(1)$, we need to obtain the disk parallelism and total disk execution time. We use the average service queue size, denoted by q, as an indicator of the disk parallelism. For the total disk execution time, we use the product of the average disk operations per second (i.e., the disk throughput), denoted by λ_d, and the average service time, denoted by s, which is obtained by the `iostat` tool. $D_d(1)$ is estimated as the total disk time divided by the disk parallelism

$$D_d(1) = \frac{\lambda_d s}{q + 1}. \tag{2}$$

Similar to the CPU, various run time disk optimizations may affect the per instance disk operations, especially under consolidation. To verify this conjecture, we measure the disk throughput with synthetic benchmarks, which write and read files to disk with a controllable intensity. Our evaluation results confirm that indeed the disk throughput depends on the amount and intensities of IO operations. In general, our synthetic benchmarks confirm that the heavier the IO workload thanks to a high number of consolidated instances, the higher the disk throughput due to run time optimizations.[4]

As an example, we show how to calculate the disk demand of n consolidated instances, $D_d(n)$ of `luindex`, a disk-intensive benchmark. The disk operations of `luindex` that are optimized (or parallelized), can be inferred from the total number of disk operations per second, o_{td}, and disk queued operations per second, o_{qd}. When $\frac{o_{qd}}{o_{td}} \simeq 1$, it implies that there is a sufficiently high workload for the disk, whereas $\frac{o_{qd}}{o_{td}} \simeq 0$ implies a low disk workload. One can expect that $D_d(n)$ may decrease from $D_d(1)$ with respect to n, which in turn indicates parallelism.

After conducting extensive empirical fitting on both synthetic micro benchmarks and `luindex`, we conclude that

[4]We do not present these results here due to lack of space.

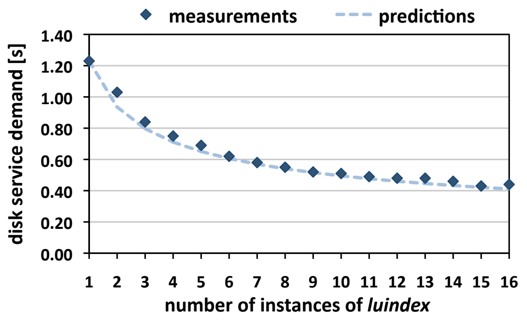


Figure 4: Disk service demand for `luindex`.

$D_d(n)$ can be well captured by $D_d(1)$ divided by n to the power of $\frac{o_{qd}}{o_{td}}$,

$$D_d(n) \quad = \quad D_d(1)/n^{o_{qd}/o_{td}}. \qquad (3)$$

Following Eq. 2 and 3, Figure 4 shows the experimentally measured and predicted disk demand of `luindex` as a function of simultaneously executing `luindex` instances.

3.3 Profiling DaCapo Benchmarks

To profile CPU and disk demands of DaCapo suite, one needs to collect the statistics required in Eq. 1- 3 and calculate the CPU saturation point. Such statistics are collected non-intrusively by standard, open source profiling tools, including `mpstat, vmstat, sar`, and `iostat`. Table 3 summarizes CPU and disk demands for a single instance ($D_c(1)$ and $D_d(1)$), cpu saturation point (ξ), number of queued disk operations per second (o_{qd}), and total number of disk operations issued per second (o_{td}).

The relative difference between $D_c(1)$ and $D_d(1)$ indicate whether an application is CPU- or disk-bound. The table shows that `luindex` is a strong disk-bound workload. However, since $D_c(n)$ and $D_d(n)$ change with the number of consolidated instances, the distinction between CPU- and disk-bound also changes as a function of the degree of consolidation. Moreover, the saturation point serves as a good indicator of the application concurrency level. High saturation points indicate a low level of concurrency; low values of ξ indicate the contrary. For example, both `batik` and `pmd` are CPU-bound workloads for any consolidated instances, but they have very different saturation points, being 7.17 and 3.85 respectively. This is because `batik`, compared to `pmd`, has a lower internal concurrency. As reported in Section 3.1.2, `pmd` is affected by a nonlinear increment of CPU demand .

3.4 Resource Demands Under Heterogeneous Consolidation

Clearly, the CPU saturation point and disk related statistics depend on the total workload required by the collocated instances and resulting run-time optimizations. As a result, in the case of heterogeneous consolidation, one needs to adjust the estimation of the CPU saturation point and the average queued and total disk operations to consider *both* types of applications. We propose to update the values based on the average of both application classes, weighted by their number of instances.

In the heterogeneous consolidation, we let the primary (secondary) application associated with subscript $i = \{1, 2\}$.

Table 3: Profiling of the DaCapo suite

benchmark	$D_c(1)$	ξ	$D_d(1)$	o_{qd}	o_{td}
`avrora`	6.88	5.52	0.12	0.0	1.9
`batik`	1.94	7.17	0.17	0.6	9.2
`fop`	0.52	5.62	0.06	0.0	18.7
`h2`	5.84	5.88	0.10	0.1	1.4
`jython`	6.01	4.30	0.65	4.7	9.4
`luindex`	1.27	8.79	1.23	63.7	161.1
`lusearch`	9.18	5.68	1.54	2.0	18.7
`pmd`	2.00	3.85	0.09	0.4	4.4
`sunflow`	15.19	6.21	0.63	0.5	5.1
`xalan`	6.09	6.49	1.05	0.6	28.2

The CPU saturation point under the consolidation is thus the average of ξ_1 and ξ_2 weighted by the number of instances of class 1 and 2, n_1 and n_2. Similar calculations are applied on o_{qd} and o_{td}. In the following, we summarize the proposed estimates , under consolidating n_1 and n_2 instances of class 1 (primary) and class 2 (secondary) applications.

$$\xi(n_1, n_2) \quad = \quad (\xi_1 \cdot n_1 + \xi_2 \cdot n_2)/(n_1 + n_2) \qquad (4)$$
$$o_{qd}(n_1, n_2) \quad = \quad (o_{qd1} \cdot n_1 + o_{qd2} \cdot n_2)/(n_1 + n_2) \qquad (5)$$
$$o_{td}(n_1, n_2) \quad = \quad (o_{td1} \cdot n_1 + o_{td2} \cdot n_2)/(n_1 + n_2). \qquad (6)$$

The CPU and disk demands can be conveniently adapted as the input for a queueing network model, which can then effectively capture the queueing and iteration times. As such, one can efficiently and accurately find the optimal consolidation for any given requirement of target iteration times. We validate our proposed profiling in conjunction with the proposed model described in the following section, by showing the difference between predicted and measured iteration times.

4. PREDICTION MODEL AND SOLVING ALGORITHM

To capture the performance interference and wait/queueing time of consolidated applications on multicores, we use a closed queueing network model which is parameterized by the proposed profiling on application CPU and disk demands. As resource demands vary according to the number of instances executed simultaneously, we develop a *load-dependent* queueing model [17]. In addition, depending on the collocated application instances being homogeneous or heterogeneous, the proposed model is correspondingly classified as single or multi-class load-dependent model.

MVA (Mean Value Analysis) [21] has been widely used to solve single and multi-class queueing networks[5]. MVA as an algorithm is quite powerful as it can provide analytical calculations of various performance metrics of interest, e.g., application throughput, iteration time, and resource utilization. To the best of our knowledge, there is no exact or approximate version of MVA that can solve a load-dependent queueing network where the resource demands depend on the collocated jobs in the system, instead the literature suggests to solve such models via simulation. Here, we enrich MVA with a new approximation algorithm that provides an analytic solution to such a load-dependent multi-class queueing networks with good accuracy.

[5]MVA provides the exact solution of product form queueing networks, whose solutions of the steady-state probabilities can be expressed as a product of factors describing the state of each queuing node.

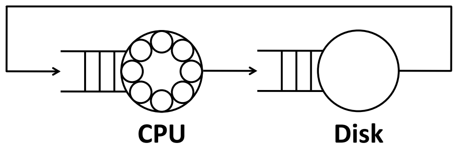


Figure 5: Two station closed queueing system. Station 1 corresponds to the aggregate of all cores and their memory, while station 2 corresponds to the disk.

4.1 Two Station Load-dependent Model

Resource demands characterize how application threads utilize the underlying resources. However, the queueing time due to performance interference among consolidated instances is yet to be predicted. Corresponding to the two most critical resources, CPU and disk, we propose a simple two station queueing system, depicted in Figure 5. Station one represents the aggregate computation capacity from available cores in the system, and station two represents a single disk. The number of consolidated application instances corresponds to the number of "jobs" in conventional closed queueing network terminology.

Assuming a closed queueing network structure as the one depicted in Figure 5, we solve the system for n simultaneous executing jobs (application instances, in our case) and estimate performance numbers for the various measures of interest. The main metric of interest here is the application iteration time, i.e., the time that one iteration completes after executing on both resources. We solve the proposed model by using multi-class load dependent MVA algorithm. The essential step is to update the resources demands according to the CPU saturation point and the disk total-queued operations according to consolidated instances.

4.2 Multi-class Load-Dependent MVA Algorithm

MVA is an algorithm that iterates through all possible states of consolidated instances that execute on resources and updates the corresponding job execution times and throughput in sequence. When homogenous consolidation is used, MVA iterates from one to n and calculates the average statistics. For the heterogeneous consolidation, MVA iterates through all possible number of jobs, $n = \{1 \ldots N\}$, in the system, and all possible combinations from jobs in each class. Furthermore, to incorporate the resource demands into the calculation of iteration times, the load-dependent MVA needs to compute the conditional probability that the CPU or the disk has j jobs of class i , provided that there are n jobs in the system. These probabilities are denoted by $P_{ki}(j|n)$, where $k = \{c, d\}$ stands for the CPU and disk respectively.

We depict a high level description of our approximated multi-class load-dependent MVA in Algorithm 1. A class is indexed by $i = \{1, 2\}$ and the resource is indexed by $k = \{c, d\}$. $T_{ci}(n_1, n_2)$ and $T_{di}(n_1, n_2)$ denote the iteration times of class i spent in CPU and disk, given n_1 and n_2 jobs in the system. The detailed description of resource demands given (n_1, n_2) consolidated instances, $D_{ci}(n_1, n_2)$ and $D_{di}(n_1, n_2)$, are described in Section 3. To ease the readability, we write $P_{ki}(j|n)$ in Algorithm 1 to encompass all the cases of $P_{ki}(j|(n_1, n_2))$ such that $n_1+n_2 = n$. We direct

Algorithm 1 Proposed MVA Approximation Algorithm

```
1: for $n$=1 to N and all combinations such that $n_1+n_2 = n$ do
2:   for class $i$=1,2 do
3:     for resource $k = c, d$ do
4:       Compute $D_{ki}(n_1, n_2)$ following Eq. (1)-(6)
5:       Compute $T_{ki}(n_1, n_2) = \sum_{j=1}^{n} D_{ki}(n_1, n_2) P_{ki}(j-1|n)$
6:     end for
7:     Compute $X(n_1, n_2) = \sum_i \frac{n_i}{\sum_k T_{ki}(n_1, n_2)}$
8:     for resource $k = c, d$ do
9:       for j= 1 to $n$ jobs at resource $k$ do
10:        Update $P_{ki}(j|n)$ [17]
11:      end for
12:    end for
13:  end for
14: end for
```

the interested reader to [17] for the detailed computation of $P_{ki}(j|n)$ in the standard Load dependent MVA algorithm. The critical step we propose here is to update the throughput $X(n_1, n_2)$ as the average of class throughput, weighted by the number of jobs in each class, i.e., $\sum_i \frac{n_i}{\sum_k T_{ki}(n_1,n_2)}$. We note that this approximation can capture the upper bound of performance measures as regulated by the bottleneck resource.

Finally, we point out that any variation in the workload/iteration time is captured via the MVA assumption that workload execution on each resource when run in isolation (i.e., no consolidation) results in exponentially distributed service demands. We will show in the following section that the exponential assumption is very effective and results in models with high prediction accuracy.

5. EVALUATION

In this section, we evaluate our proposed methodology to predict iteration times and to identify optimal homogeneous and heterogeneous consolidations. The detailed description of the DaCapo benchmarks and the reference system can be found in Section 2. The profiling interval has a duration of 5 minutes, including 2 minutes of warm-up phase, and the profiling results are summarized in Section 3.3.

5.1 Homogeneous Consolidation

In this subsection, we search for optimal homogeneous consolidations of 10 DaCapo benchmarks using the proposed methodology and validate our predictions with experimental data. We first summarize the predicted and measured iteration times in Figure 6. In more than half of the benchmarks, the iteration time increases very slowly until the number of consolidated instances reaches the saturation point ξ for the specific workload, as listed in Table 1. On the contrary, `avrora` and `pmd` show more increase in their iteration times, even for a small number of consolidated instances. This behavior is due to CPU run-time optimizations that we model via the load-dependent methodology. Also `luindex` exhibits a similar trend, due to disk run-time optimizations. Our predictions characterize those trends very well and capture the increments in iteration times, even the saturation point.

The average error computed from all consolidations is around 6%. The highest prediction errors are 21% for con-

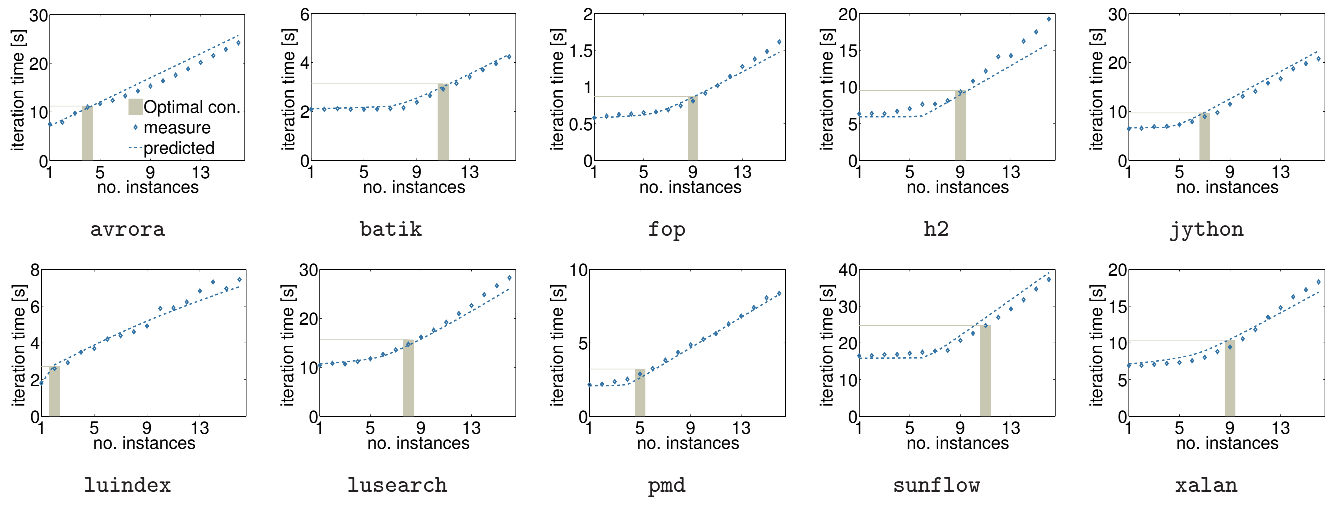


Figure 6: Prediction of iteration times for homogeneous consolidation of all DaCapo benchmarks.

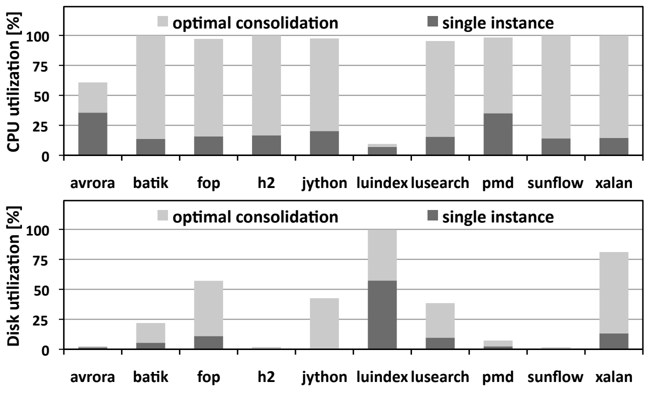


Figure 7: CPU and disk utilization of optimal homogeneous consolidations and single instance execution.

solidating 6 instances of `h2` and 17% when consolidating 16 instances of `h2`. Overall, when the number of consolidated instances is small, i.e., less than 10, prediction errors are small. These configurations are particularly interesting to service providers, as the system is not yet saturated. Our proposed methodology is particularly accurate in predicting iteration times for different consolidations, and also provides accurate results when the system is saturated. Overall, Figure 6 confirms the precision of our predictions for both CPU and disk intensive applications.

5.1.1 *Optimal Consolidation of DaCapo*

As benchmarks have a wide variety of resource demands, we define the target iteration time for all benchmarks as 1.5 times the iteration time of single instance, i.e., $1.5 \cdot T(1)$.[6] Using our predicted values and given a performance target for the average iteration, one can easily identify the optimal number of consolidated instances that achieve this performance target. In Figure 6 we highlight optimal instances and the corresponding target iteration times – see the gray column and the corresponding target value (on the y-axis) on each graph. In many cases, increasing the number of consolidated instances does not have an impact on the iteration time. This is mainly due to the intrinsic parallelism offered by the CPU: as long as the CPU is not saturated, additional workload instances can be executed in parallel without affecting the iteration time. When the CPU saturates, the iteration time increases with the number of consolidated instances but there is no clear rule of thumb that one could use to project iteration times when more applications are consolidated. Overall, we see that the model consistently predicts the optimal consolidation given a target iteration time across workloads with very different characteristics.

Figure 7 summarizes CPU and disk utilization under the optimal consolidation and under single instance execution. For the optimal consolidation experiments, we used the target performance of $1.5 \cdot T(1)$ for iteration times. Resource utilization and system throughput are tremendously improved after consolidation. However, as most benchmarks are either CPU- or disk-intensive, optimal homogeneous consolidations often result in unbalanced resource utilizations. However, the degree of unbalance of CPU and disk usage in the single instance case does not necessarily carry in the consolidated cases, see `batik` and `xalan`.

5.1.2 *SPECjvm2008 Benchmarks*

To further test the applicability of our methodology on different workloads, i.e., beyond DaCapo benchmarks, we consider consolidations of benchmarks from the SPECjvm2008 suite[7]. In particular, we consider homogeneous consolidations of 2, 4, 8, and 16 instances of each benchmark in the suite, excluding only the benchmark `compiler` (due to runtime exceptions when multiple instances of `compiler` are concurrently executed on the same machine). Over a total of 76 considered consolidations, the average prediction error is 8.9%, comparable to the measured error for the DaCapo benchmark suite. Because of space limitations, we do not present the detailed results for the experiments with SPECjvm2008.

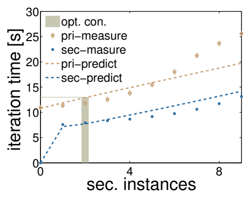
[6]Of course, any alternative definition of a "target" iteration would also work.

[7]See http://www.spec.org/jvm2008

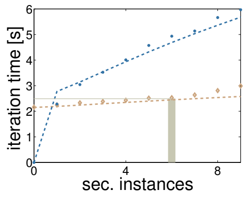
Table 4: Prediction error [%] for iteration time under various heterogenous consolidations

	Primary application																			
	avrora				batik				fop				h2				jython			
Secondary app.	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4
avrora	3	7	9	10	16	15	19	23	13	7	11	14	22	14	12	15	5	18	30	32
batik	6	12	18	24	4	4	5	9	6	6	5	7	12	13	9	8	1	6	11	13
fop	5	6	11	14	4	4	5	7	5	2	3	5	10	12	9	4	4	6	6	6
h2	6	7	11	10	3	3	7	5	8	8	6	4	16	21	9	2	1	1	7	8
jython	3	8	20	37	5	3	10	12	7	6	7	9	13	5	8	8	1	8	10	15
luindex	7	7	8	10	6	4	2	3	10	8	11	11	8	10	13	12	5	5	4	5
lusearch	2	3	3	7	3	6	10	15	4	4	8	11	12	14	8	13	9	16	16	20
pmd	4	15	25	35	2	3	3	3	9	5	9	13	16	6	8	8	4	4	4	8
sunflow	7	8	11	14	3	6	8	8	7	6	5	4	16	16	10	2	6	7	5	13
xalan	5	7	7	9	3	3	4	6	4	2	2	5	11	13	9	8	10	13	7	5

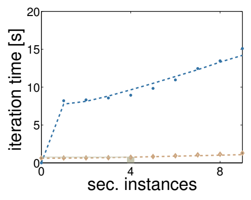
	luindex				lusearch				pmd				sunflow				xalan			
Secondary app.	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4	4;1	4;2	4;3	4;4
avrora	6	8	6	6	9	7	6	8	8	22	31	38	17	14	12	15	11	8	7	9
batik	4	2	4	4	2	4	10	14	5	5	3	2	6	6	8	8	6	6	5	6
fop	3	5	6	8	3	6	8	11	6	6	8	12	7	9	4	6	4	3	3	4
h2	1	6	6	12	4	10	10	10	6	8	4	11	8	12	6	4	3	5	7	10
jython	12	2	8	8	8	11	18	20	3	3	4	6	10	6	9	14	3	6	8	6
luindex	5	1	4	6	17	16	17	13	13	15	18	21	10	6	4	5	10	16	19	23
lusearch	4	7	10	14	0	3	2	2	11	15	22	23	6	7	3	5	8	9	12	14
pmd	5	9	13	19	11	16	22	22	9	4	5	5	12	8	9	7	7	17	22	19
sunflow	2	0	2	2	1	4	4	7	6	7	8	6	7	9	2	9	5	4	4	5
xalan	15	18	20	23	4	7	10	14	14	17	22	23	5	6	3	5	9	10	11	10



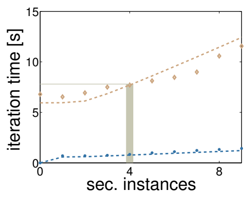
(1) `avrora+xalan`



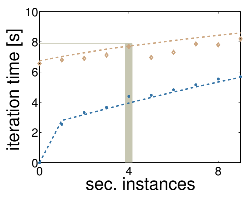
(2) `batik+luindex`



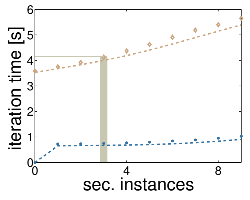
(3) `fop+xalan`



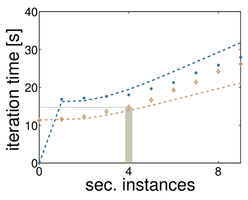
(4) `h2+fop`



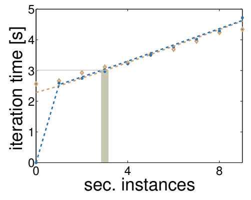
(5) `jython+luindex`



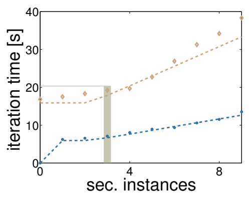
(6) `luindex+fop`



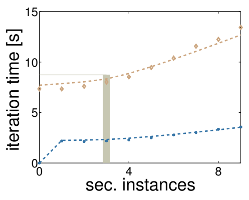
(7) `lusearch+sunflow`



(8) `pmd +batik`



(9) `sunflow+h2`



(10) `xalan+batik`

Figure 8: Two-class heterogeneous consolidation (`primary+secondary`): iteration times when consolidating 4 primary application instances and a varying number of secondary application instances.

5.2 Two-Class Heterogeneous Consolidation

In this subsection, we evaluate the prediction errors of iteration times of primary and secondary applications in various two-class heterogeneous consolidations. We consider the following scenario: one application is considered to be of the primary type. For this application, four instances should be executing simultaneously.[8] The four primary applications are consolidated with a secondary application whose instances vary from 1 to 4.

For each of the 10 benchmarks, we select one as the primary application and pair it with all other nine benchmarks that are considered as the secondary one. Table 4 summarizes the relative prediction errors, presented as the average of prediction errors from the primary and the secondary applications. Overall, the average prediction error is only 8.1% across 400 possible heterogeneous consolidations. Note that this extensive experimentation to find the ideal pairing of primary and secondary applications can be very time consuming and could grow exponentially with the number and combination of consolidated instances. With our proposed methodology, we require only *a limited number* of profiling runs of single instances and achieve very accurate predictions across a large number of consolidations.

Prediction errors are generally higher for large numbers of secondary application instances, because either the CPU or disk is over-saturated. The pairing of applications with the highest errors are `avrora` and `pmd`. Recall the discussion in Section 3.1.2, both `avrora` and `pmd` make extensive use of spin locks and their performance highly depends on the number of collocated instances.

[8]Of course, this scenario can be changed and the number of primary execution instances can be any integer. We have done experiments that have varied this number from 1 to 10 but are not reported here due to lack of space. The selected number of four primary consolidated applications is representative of all experiments.

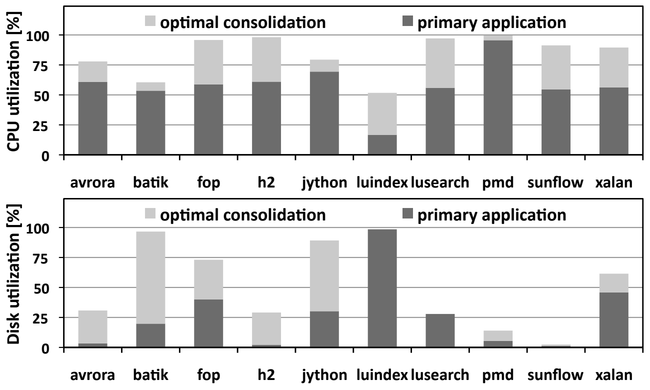


Figure 9: CPU and disk utilization of ten heterogeneous consolidations.

5.2.1 Optimal Consolidation

In this subsection, we define the goal that an optimal two-class heterogeneous consolidation corresponds to the maximum number of instances of the secondary application such that the overhead factor of the four primary application instances is less than factor 1.2.

In Figure 8, we illustrate the predicted and measured iteration times for ten cases of heterogeneous consolidation. With respect to the target iteration times, we identify the optimal number of instances of the secondary application using our prediction. We summarize the optimal heterogeneous consolidation in terms of the secondary application and the corresponding number of instances in Figure 8. The corresponding CPU and disk utilization values are also illustrated in Figure 9.

In case (2), `batik + luindex`, we can accurately predict iteration times for both benchmarks. The maximum number of consolidated instances of `luindex` without violating the target iteration time of `batik` is 6. As `luindex` is disk intensive and `batik` is CPU intensive, the optimal consolidation also increases both CPU and disk utilization. Case (5), `jython + luindex`, and case (6), `luindex + fop`, also benefit from complementary resource usage patterns. In most of the cases where two CPU-bound benchmarks are consolidated, the optimal number of secondary application instances is three or four, and the resulting CPU utilization is more than 95%.

5.3 Three-Class Heterogeneous Consolidation

In this subsection, we evaluate our predictions with various three-class heterogeneous consolidations. In particular, we consider only those cases in which the number of consolidated instances is the same for each class. A total of 360 three-class consolidation is evaluated. The aim here is to demonstrate our methodology can be applied on a higher degree of consolidation, i.e., three and even higher number of classes, and provide accurate prediction on iteration times of each class. Consequently, we do not stress the difference of target iteration times among primary and secondary applications, nor the optimal consolidation.

Figure 10 illustrates a small selection of the considered consolidations, in which the number of consolidated units goes from 1 to 3 for each class. As can be seen from the figures, our methodology accurately predicts the iteration

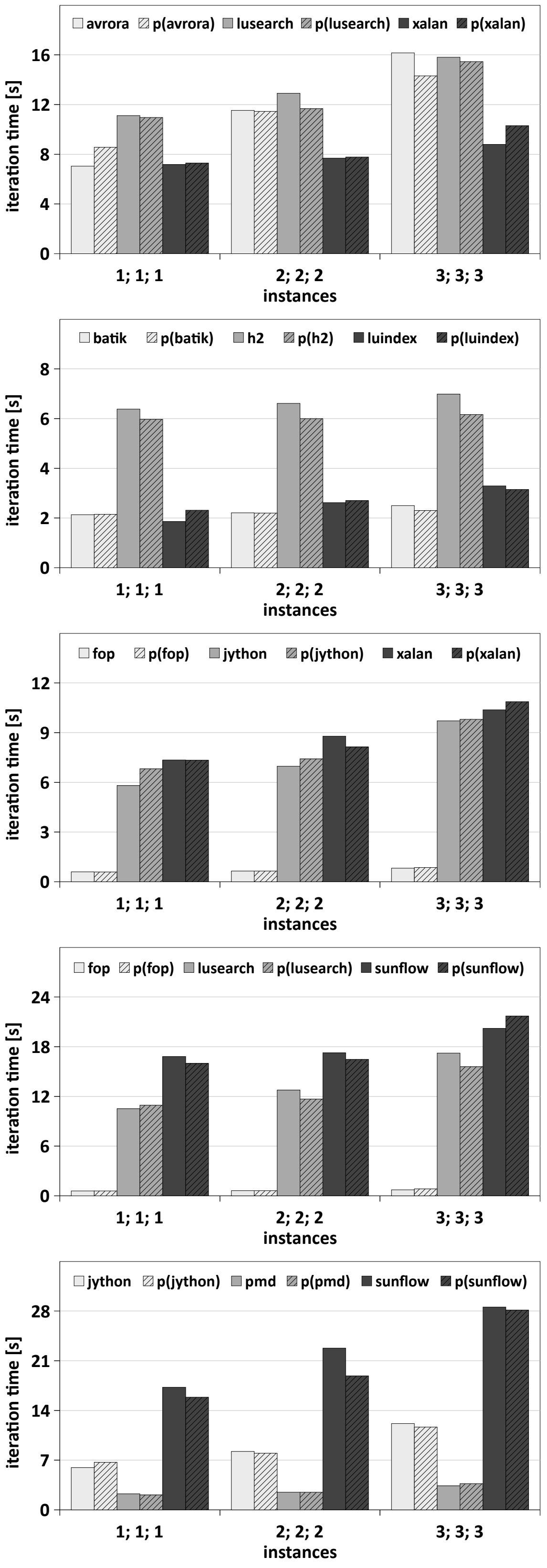


Figure 10: Three-class heterogeneous consolidation: measured v.s. predicted iteration times when consolidating different instances of applications. The notation "p" represents the predicted value.

time overhead for each class of consolidated applications. Over a total of 360 unique consolidations, the average error, computed from all three classes, is only 10.43%. This exhaustive evaluation confirms the stability of our results and suggests that our methodology can provide sufficiently accurate predictions for a higher number of consolidated classes of workloads.

6. DISCUSSION

Using our profiling and modeling methodology, we demonstrated that we can accurately predict iteration times for an extensive set of homogeneous or heterogeneous consolidations, and on a different benchmark suites, i.e., micro benchmarks, DaCapo and SPECjvm2008. Experiments show how our approach accurately predicts the target iteration times of a primary application. The same analysis and model can be used to predict the target iteration times of a secondary application. We stress that our model can be used to target different metrics, e.g., the minimal iteration slow-downs of the secondary application, to identify the optimal secondary application, to identify consolidations that meet certain system throughput or utilization levels.

In this paper, we mainly focused on a single example: how to best predict the iteration time of the primary application. Frequently, the optimal consolidation of primary with secondary applications is to achieve equal utilization across *all* system resources such that the system throughput and utilization are optimized. We would like to stress that the optimal heterogeneous consolidation indeed varies according to the definition of performance objectives. When the performance objective is to maintain a target weighted iteration time of two classes, it might be preferable to consolidate two CPU-bound applications with different concurrency levels. Such counter-intuitive solutions cannot be easily identified with existing profiling tools. Our solution not only captures the dynamics of consolidated workload but also provides a robust solution for performance prediction of a wide range of objectives.

7. RELATED WORK

Various simulation-based and model-based approaches have been developed to address workload performance interference on multicore systems. In particular, the performance metrics of interest are usually shared hardware counters, such as cycle per instruction (CPI) and L2/L3 cache misses. In [12], the authors use artificial neural networks to predict IPC for exploring the architecture design space and validate their results via simulation. Each benchmark is profiled, validated, and predicted in isolation. Chen et al. [6] estimate multi-resource demands, e.g., L2 cache size and memory bandwidth, of a single application, using ILP and MLP models, and validate their models via simulation. These models are not validated on consolidated workloads. In contrast, Lee et al. [16] propose composable regression models to predict CPI delay, considering consolidating multiple classes of application instances. METE [22] predicts the end-to-end IPC of collocating two benchmarks via simulation. The objective of METE is to dynamically provide sufficient on-chip resources to applications such that the target IPC is achieved via a feedback control approach.

The main focus of profiling single- and multi-threaded applications on multicore systems is to characterize the performance interference on hardware resources, e.g., cache misses and thread contention. Most characterization analysis considers a single instance of a single-threaded application and a few are applied on consolidated instances of applications [13, 8, 4]. The profiling approach can be non-intrusive using existing OS counters [14, 26, 27] or require modification of applications' source code or bytecode [10, 24]. Zhuravlev et al. [28] characterize resource contention of multi-threaded applications in the memory hierarchy, but their main focus is on comparing the systems while executing the application in isolation. In [4], the authors characterize the slow down of core interference, cache interference and virtual overhead, due to consolidating workloads on a multicore dual-processor Intel platform. Dey et al. [8] propose a general methodology to characterize any multi-threaded application for the last level cache contention, and private cache contention.

To summarize, existing methodologies combine model and simulation to address system-centric performance targets, whereas our proposed methodology can accurately and efficiently predict user-centric targets, i.e., iteration times, while maximizing performance targets, such as resource utilization, on a real system. In addition, our methodology predicts optimal consolidations by using only light-weight and non-intrusive profiling. We also stress that the proposed methodology can be used for consolidation predictions of any performance measure, we direct the interested reader to [2] for results that show optimal consolidations of even conflicting performance measures, and specifically when the target is the ratio of system throughput (that is aimed to be maximized) over the application execution time (that is aimed to be minimized).

8. CONCLUSION

In this paper, we develop a light-weight and non-intrusive methodology to achieve application-centric performance targets, for consolidating homogeneous and heterogeneous application instances on modern multicore systems. Using a very small number of profiling runs, our proposed profiling encapsulates crucial load dependent characteristics of applications, i.e., resource demands, application concurrency level, hardware parallelism, and the impact of run time optimization schemes. Furthermore, based on the resource demands profiled, we develop a load-dependent queueing model to predict various performance metrics. Our evaluation of consolidating multithreaded benchmarks with different concurrency levels on an IBM Power7 system shows a prediction error below 10 % over more than 900 consolidation cases.

Overall, our proposed methodology can accurately and efficiently achieve optimal consolidation, for any given combination of target iteration times of heterogeneous applications, in particular Java workloads. As our proposed methodology relies on low-level and easily-accessed performance counters, the methodology readily applies to more complex workloads. In our future work we intend to focus on network-intensive applications (i.e., extend the model by providing an additional queueing station) and on applying the methodology on different hardware platforms. We are also working on a prediction tool that focuses on automating the process of finding the best consolidation matches on multicores [1, 3].

Acknowledgments

This work has been supported by IBM and the Swiss National Science Foundation. Part of this work was conducted while Danilo Ansaloni was on an internship and Evgenia Smirni was on sabbatical leave at the IBM Zürich Research Laboratory. Evgenia Smirni is partially supported by NSF grants CCF-0811417 and CCF-0937925.

9. REFERENCES

[1] D. Ansaloni, L. Y. Chen, E. Smirni, and W. Binder. Towards autonomic consolidation of heterogeneous workloads. In *Workshop on Posters and Demos Track*, Middleware, pages 12:1–12:2, 2011.

[2] D. Ansaloni, L. Y. Chen, E. Smirni, and W. Binder. Model-driven Consolidation of Java Workloads on Multicores. In *Proceedings of DSN-PDS*, 2012.

[3] D. Ansaloni, L. Y. Chen, E. Smirni, A. Yokokawa, and W. Binder. Find Your Best Match: Predicting Performance of Consolidated Workloads. In *ICPE 2012 Posters/Demos Track*, ICPE, 2012.

[4] P. Apparao, R. Iyer, X. Zhang, D. Newell, and T. Adelmeyer. Characterization & Analysis of a Server Consolidation Benchmark. In *Proceedings of VEE*, pages 21–30, 2008.

[5] 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*, pages 169–190, 2006.

[6] J. Chen, L. John, and D. Kaseridis. Modeling Program Resource Demand Using Inherent Program Characteristics. In *Proceedings of SIGMETRICS*, pages 1–12, 2011.

[7] L. Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, R. Harper, and B. Morris. Consolidating Clients on Back-end Servers with Co-location and Frequency Control. *SIGMETRICS Perform. Eval. Rev.*, 34:383–384, June 2006.

[8] T. Dey, W. Wang, J. Davidson, and M. Soffa. Characterizing Multi-threaded Applications Based on Shared-resource Contention. In *Proceedings of ISPASS*, pages 76–86, 2011.

[9] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of SOCC*, 2011.

[10] M. Hauswirth, P. Sweeney, A. Diwan, and M. Hind. Vertical Profiling: Understanding the Behavior of Object-oriented Applications. In *Proceedings of OOPSLA*, pages 251–269, 2004.

[11] M. R. Hines, A. Gordon, M. Silva, D. da Silva, K. D. Ryu, and M. Ben-Yehuda. Applications Know Best: Performance-Driven Memory Overcommit With Ginkgo. Technical report, IBM, 2011.

[12] E. İpek, S. McKee, R. Caruana, B. de Supinski, and M. Schulz. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proceedings of ASPLOS*, pages 195–206, 2006.

[13] N. Jerger, D. Vantreaseand, and M. Lipast. An Evaluation of Server Consolidation Workloads for Multi-Core Designs. In *Proceedings of IISWC*, pages 47–56, 2007.

[14] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28:54–66, 2008.

[15] 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.

[16] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable Performance Regression for Scalable Multiprocessor Models. In *Proceedings of Micro*, pages 270–281, 2008.

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

[18] 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 ICAC*, pages 11–20, 2010.

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

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

[21] M. Reiser and S. S. Lavenberg. Mean-Value Analysis of Closed Multichain Queuing Networks. *J. ACM*, 27:313–322, 1980.

[22] A. Sharifi, S. Srikantaiah, A. Mishra, M. Kandemir, and C. Das. METE: Meeting End-to-end QoS in Multicores Through System-wide Resource Management. In *Proceedings of SIGMETRICS*.

[23] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A Case for Scaling Applications to Many-core with OS Clustering. In *Proceedings of EuroSys*, pages 61–76, 2011.

[24] N. Tallent and J. Mellor-Crummey. Effective Performance Measurement and Analysis of Multithreaded Applications. *SIGPLAN Not.*, 44:229–240, 2009.

[25] 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 SIGMETRICS*, pages 291–302, 2005.

[26] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proceedings of Middleware*, pages 366–387, 2008.

[27] 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.

[28] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Proceedings of ASPLOS*, pages 129–142, 2010.