



Predicting Application Resource Requirements in Virtual Environments

Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, Prashant Shenoy

HP Laboratories
HPL-2008-122

Keyword(s):

virtualization, application resource usage, benchmarking, modeling, automation, performance models, regression-based approach

Abstract:

Next Generation Data Centers (NGDC) are transforming labor-intensive, hard-coded, siloed systems into shared, virtualized, automated, and fully managed adaptive infrastructures. Virtualization technologies promise great opportunities for reducing energy and hardware costs through server consolidation. Moreover, virtualization can optimize resource sharing among applications hosted in different virtual machines to better meet their resource needs. However, to safely transition an application running natively on real hardware to a virtualized environment, one needs to estimate the additional resource requirements incurred by virtualization overheads. In this work, we design a general approach for estimating the resource requirements of applications when they are transferred to a virtual environment. Our approach has two key components: a set of microbenchmarks to profile the different types of virtualization overhead on a given platform, and a regression-based model that maps the native system usage profile into a virtualized one. This derived model can be used for estimating resource requirements of any application to be virtualized on a given platform. Our approach aims to eliminate error-prone manual processes and presents a fully automated solution. We illustrate the effectiveness of our methodology using Xen virtual machine monitor. Our evaluation shows that our automated model generation procedure effectively characterizes the different virtualization overheads of two diverse hardware platforms and that the models have median prediction error of less than 5% for both the RUBiS and TPC-W benchmarks.



Predicting Application Resource Requirements in Virtual Environments^{*}

Timothy Wood¹, Ludmila Cherkasova², Kivanc Ozonat², and Prashant Shenoy¹

¹ University of Massachusetts, Amherst, {twood, shenoy}@cs.umass.edu

² HP Labs, Palo Alto, {lucy.cherkasova, kivanc.ozonat}@hp.com

Abstract

Next Generation Data Centers (NGDC) are transforming labor-intensive, hard-coded, siloed systems into shared, virtualized, automated, and fully managed adaptive infrastructures. Virtualization technologies promise great opportunities for reducing energy and hardware costs through server consolidation. Moreover, virtualization can optimize resource sharing among applications hosted in different virtual machines to better meet their resource needs. However, to safely transition an application running natively on real hardware to a virtualized environment, one needs to estimate the additional resource requirements incurred by virtualization overheads.

In this work, we design a general approach for estimating the resource requirements of applications when they are transferred to a virtual environment. Our approach has two key components: a set of microbenchmarks to profile the different types of virtualization overhead on a given platform, and a regression-based model that maps the native system usage profile into a virtualized one. This derived model can be used for estimating resource requirements of any application to be virtualized on a given platform. Our approach aims to eliminate error-prone manual processes and presents a fully automated solution. We illustrate the effectiveness of our methodology using Xen virtual machine monitor. Our evaluation shows that our automated model generation procedure effectively characterizes the different virtualization overheads of two diverse hardware platforms and that the models have median prediction error of less than 5% for both the RUBiS and TPC-W benchmarks.

1 Introduction

Virtualization and automation are key capabilities of Next Generation Data Centers (NGDC). The primary motivation for enterprises to adopt virtualization technologies is the promise of creating a more agile and dynamic IT infrastructure. Virtualization separates the hardware owner from the application owner – allowing system configuration, monitoring, and management to be homogenized and automated across the data center. While masking the details of server resources from users, virtualization can optimize resource sharing among applications hosted in different virtual machines via the ability to quickly repurpose server capacity on demand, and hence better meet the needs of applications and respond more effectively to changing business conditions.

In NGDC, where server virtualization provides the ability to slice larger, underutilized physical servers into smaller, virtual ones, fast and accurate *performance models* become instrumental for enabling applications to be consolidated, optimally placed and

^{*} This work was largely completed in the summer of 2007 when Tim Wood did an internship at HPLabs. A shorter version of this paper will appear in Middleware'2008.

provided with the necessary resources. In order to evaluate which workloads can be consolidated to which servers, some capacity planning and workload analysis must be done. In the simple naive case, the service provider may estimate the peak resource requirements of each workload and then evaluate the combined resource requirements of a group of workload by using the sum of their peak demands. However, such an approach can lead to significant resource over-provisioning since it does not take into account the benefits of resource sharing to accommodate the complementary workload patterns. A more promising and accurate approach for the design of workload placement services employs a trace-based approach that assesses permutations and combinations of workload patterns in order to determine the optimal stacking functions [27, 29, 10]. Under this approach, the application resource usage traces are routinely collected over some time period (typically 3-6 months) to get a representative application resource usage profile. Then these traces are used for capacity planning and workload placement in workload consolidation exercises (see existing commercial tools [14, 33, 37]). The general idea behind trace-based methods is that the historic traces that capture past application demands are representative of the future application behavior.

However, capacity planning when transitioning to a virtual environment poses additional challenges due to overheads caused by the virtualization layer. These virtualization overheads depend on the type and implementation specifics of the virtualization solution [31, 40, 18, 5]. Often, the “amount” of CPU overhead is directly proportional to the “amount” of performed I/O processing [7, 11]. Current trace-based capacity planning and management solutions have the capability to scale workload traces by a specified CPU-multiplier to account for hardware changes between platforms, but this form of scaling may not be effective when moving to a virtualized platform which can exhibit very different levels of overhead depending on the rate and type of I/O being performed by an application.

In this work, we design a general approach for estimating the CPU requirements of applications when they are transferred to a virtual environment. While the application’s requirements for network and disk traffic do not change, the amount and type of I/O might impact the amount of CPU required by the application.

Our approach has the following key components:

- A selected set of microbenchmarks to profile the different types of virtualization overhead on a given platform. This microbenchmark suite is executed on the native hardware and in a virtualized environment to create two resource usage profiles: *i) native* and *ii) virtualized*;
- Using a regression-based approach we create a model that maps the *native* system usage profile into the *virtualized* one. This model helps to predict the resource requirements of any application to be virtualized on a given platform.

The correct execution phase of the microbenchmark suite is a prerequisite for building an accurate model between native and virtualized platforms. If some microbenchmarks have malfunctioned or collected data were corrupted then it can inevitably impact the model outcome. To evaluate the quality of experimental data and automate the overall process, we perform an additional analysis to filter out microbenchmark data with high error against the obtained regression-based model. Then, a more accurate model

is created by using the reduced data set. We also can rerun identified “failed” or “malfunctioned” microbenchmarks and repeat the analysis phase. Such an approach aims to eliminate error-prone manual processes in order to support a fully automated solution.

We illustrate the effectiveness of our methodology using Xen virtual machine monitor [5, 41]. The evaluation shows that our automated model generation procedure effectively characterizes the different virtualization overheads of two diverse hardware platforms and that the models have a median prediction error of less than 5% for both the RUBiS [3] and TPC-W [34] benchmarks.

The rest of this paper is organized as follows. Section 2 provides the problem definition. Section 3 presents the suite of microbenchmarks used for platform profiling and explains our objectives for their selection. Section 4 introduces our regression-based model for predicting application resource requirements in virtual environments. Section 5 evaluate the effectiveness of our models for several realistic applications and different systems. Section 6 discuss challenges and directions for future work. Section 7 presents related work. Finally, a summary and conclusions are given in Section 8.

2 Problem Definition

Server consolidation is an approach to reduce the total number of servers in response to the problem of server sprawl, a situation in which multiple, under-utilized servers take up more space and consume more resources than can be justified by their workloads. *Virtual Machine Monitors (VMMs)* enable diverse applications to run in isolated environments on a shared hardware platform, and provide a degree of fault and performance isolation between the applications.

A typical approach for evaluating which workloads can be efficiently consolidated together is based on multi-dimensional “binpacking” of resource usage traces. Under such an approach, each application is characterized by its CPU, I/O and memory usage over time. Then a binpacking algorithm finds a combination of workloads with resource requirements which do not exceed the available server resources. After the initial workload placement, specialized workload management tools are used [15, 13] to dynamically adjust system resources to support the required application performance.

In our work, we are concerned with the initial workload placement phase that requires as an input the application resource usage traces in virtual environment. Resource requirements (in particular, CPU requirements) can increase due to virtualization overheads. It is important to know what an application’s resource needs are going to be prior to transitioning it to the virtual environment. If these overheads are not accounted for during initial planning, an application could be deployed to a server with insufficient resources, resulting in unacceptable application performance.

Xen and VMware ESX server demonstrate the two popular I/O models for VMs. In ESX (and Xen in its original design [5]), the hypervisor itself contains device driver code and provides safe, shared access for I/O hardware (see Figure 1 a). Later, the Xen team proposed a new architecture [9] that allows unmodified device drivers to be hosted and executed in isolated “driver domains” (see Figure 1 b).

In Xen, the management domain Dom-0 hosts unmodified Linux device drivers and plays the role of the driver domain. This I/O model results in a more complex CPU

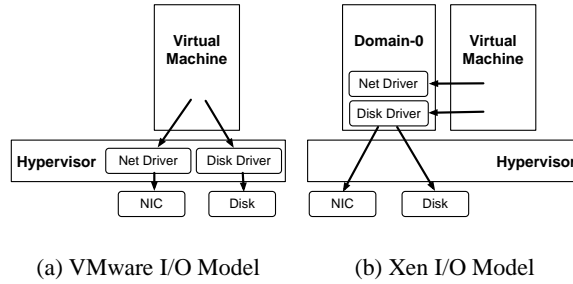


Fig. 1. Two popular I/O models for VMs.

usage model. For I/O intensive applications, CPU usage has two components: CPU consumed by the guest virtual machine (VM) and CPU consumed by Dom-0 which performs I/O processing on behalf of the guest domain.

In this work, without loss of generality, we demonstrate our approach using Xen running paravirtualized VMs. We believe that our approach can be applied to other virtualization platforms such as VMware ESX Server, but focus on Xen in this work because it presents the additional challenge of modeling both the virtualized application and the driver domain (Dom-0) separately.

Given resource utilization traces of an application running natively, we aim to estimate what its resource requirements would be if the application were transitioned to a virtual environment on a given hardware platform. For example, let a collection of application resource usage profiles (over time) in native system be provided as shown in Figure 2 (top): *i*) CPU utilization, *ii*) transferred and received networking packets, *iii*) read and written disk blocks.

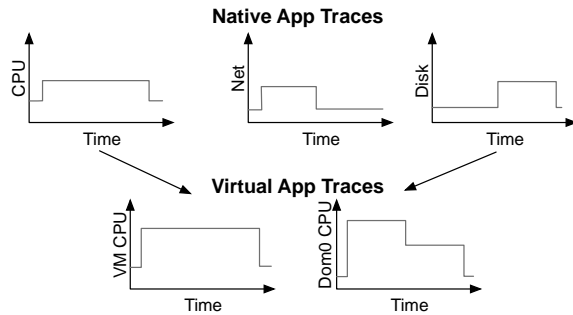


Fig. 2. Using native application traces to predict resource needs in virtual environments.

The *goal* is to estimate the CPU requirements of the following two components as shown in Figure 2 (bottom):

- virtual machine (VM) where the application is going to reside and execute;
- Dom-0 which performs I/O processing on behalf of the guest virtual machine.

Intuitively, we expect that CPU utilization of VM is highly correlated and proportional to the native CPU usage profile of the application, while Dom-0 CPU utilization is mostly determined by a combination of I/O profiles (both network and disk).

We focus on estimating only CPU utilization since other metrics (such as disk and network request rates) are not directly impacted by the virtualization layer—running an application in a virtualized environment will not cause more packets to be sent over the network or more disk requests to be generated. Instead, the virtualization layer incurs additional processing overheads when I/O is performed; it is these overheads which our models seek to capture.³

Our Approach: We present an automated model generation system which determines the relationship between the native and virtual platforms being used. The overhead of the virtual platform is characterized by running a series of microbenchmarks on both platforms and building a model that relates the resource requirements on one platform to the other. Although it is created using data from synthetic benchmarks, the result is a general model which can be applied to traces from any other application in order to predict what its resource requirements will be on the virtual platform.

3 Platform Profiling

In this section, we describe the collection of microbenchmarks that are selected for profiling different types of virtualization overhead on a given platform. In order to determine a general relationship between the application resource usage in native and virtual platforms, we first accumulate the samples of such usage profiles by executing a specially selected set of microbenchmarks in both native and virtualized environments.

3.1 Microbenchmark Requirements

The microbenchmark selection for our suite is driven by the following objectives:

- *Microbenchmarks must be capable of applying a range of workload intensities.*

There are a large number of benchmarks available which allow you to stress test a system to see how it performs under maximum load. However, a typical enterprise application exhibits variable workloads. A benchmark which simply reports the maximum number of web requests or disk accesses that a system can perform per second is not useful for us since it only provides information about the maximum capacity and corresponding resource usage, not about the utilization under different workloads. In consolidation scenarios, the considered applications are likely to operate at a light or medium load. Therefore, we concentrate on creating a suite of microbenchmarks that can be configured to generate workloads of different *intensities*, i.e., capable of generating different networking/disk access rates as well as consume different CPU amounts.

- *The same microbenchmark should run nearly-identical in both native and virtual environments.*

This requirement is very important for our approach. The application behavior is represented via different resource usage traces over time. When a workload performs a combination of CPU and I/O activities at time interval T on a native system, we correlate it with the CPU usage profile (both VM and Dom-0) observed at time interval T

³ Virtualization also incurs a memory overhead. Both Xen and ESX Server require a base allocation for Dom-0 or the Service Console, plus a variable amount per VM.

in the virtualized environment for the same workload in order to build the model (relationship) between the native and virtualized systems. Thus, the requirement for our microbenchmarks is that the workloads must be *nearly-identical* in both the native and virtual environments we test. While our benchmarks allow some non-determinism in the workload traffic patterns, we carefully design our microbenchmarks to always execute the same set of activities over the same period of time. We avoid benchmarks with a strong feedback loop since virtualization overheads may increase latency and distort the resource usage over time. ⁴

3.2 Microbenchmark Workloads

The selected microbenchmarks have to create a set of workloads that utilize different system resources and have a different range of workload intensities.

We use a client-server style setup in our benchmarks. In general, a client machine issues a set of requests to the benchmark server running on the system being profiled. The clients adjust the rate and type of requests to control the amount of CPU computation and I/O activities performed on the test system.

At a high level, our microbenchmarks are comprised of three basic workload patterns that either cause the system to perform CPU intensive computation, send and receive network packets, or read and write to disk.

- Our *computation intensive* workload calculates Fibonacci series when it receives a request. The number of terms in the series can be varied to adjust the computation time.
- The *network intensive* workload has two modes depending on the type of request. In transmit mode, each incoming request results in a large file being sent from the system being tested to the client. In receive mode, the clients upload files to the benchmark application. The size of transferred files and the rate of requests is varied to adjust the network utilization rate.
- The *disk intensive* workload has read and write modes. In both cases, a random file is either read from or written to a multilevel directory structure. File size and request rate can be adjusted to control the disk I/O rate.

Each workload is created by adjusting the request type sent to the server from the client machines. We split each of the basic benchmark types, CPU-, network-, and disk-intensive, into five different intensities ranging from 10% load to 90% load. The maximum load that a server can handle is determined by increasing the throughput of benchmark requests until either the virtual machine or Dom-0 CPU becomes saturated during testing. It is important that a range of intensities be measured for each benchmark type in order to ensure we can build an accurate model for the full working range of the virtual server.

To create more complex and realistic scenarios, we use a *combination* workload that exercises all three of the above components. The combination workload simultaneously sends requests of all types to the benchmarked server. The relative intensity of each

⁴ Section 6 provides a more detailed discussion on the issue of “applications with a feedback loop”.

request type is varied in order to provide more realistic training data which does not focus exclusively on a single form of I/O.

The microbenchmarks are implemented as a set of PHP scripts running on an Apache web server at the benchmarked server side. Basing the microbenchmarks on Apache and PHP has the benefit that they can be easily deployed and executed on a wide range of hardware platforms within a software environment which data center administrators are already familiar with. The developed microbenchmark suite allows us to generate a diverse set of simple and more complex workloads that exercise different system components. The full set of PHP scripts, as well as the scripts to create the file structure used in the disk tests, comprise only a few hundred lines of code.

The client workloads are generated using *httperf* [23] and Apache JMeter [4]. These tools provide flexible facilities for generating variable and fixed rate HTTP workloads. The workloads can then be easily “replayed” in different environments. Both tools can emulate an arbitrary number of clients accessing files on a webserver; we use JMeter primarily because it supports uploading files as part of the client interactions.

3.3 Platform Resource Usage Profiles

We generate *platform profiles* by running a set of microbenchmarks on the systems being tested. While each microbenchmark is running, we gather resource utilization traces to define the platform profile used as the training data for the model. Within the native system, we currently gather information about *eleven* different resource metrics related to CPU utilization, network activity, and disk I/O. The full list of metrics is shown in Table 1. These statistics can all be gathered easily in Linux with the *sysstat* monitoring package [32]. We focus on this set of resource measurements since they can easily be gathered with low overhead. Since these traces must also be gathered from the live application being transitioned to the virtual environment, it is crucial that a lightweight monitoring system can be used to gather data.

CPU	Network	Disk
User Space %	Rx packets/sec	Read req/sec
Kernel %	Tx packets/sec	Write req/sec
IO Wait %	Rx bytes/sec	Read blocks/sec
	TX bytes/sec	Write blocks/sec

Table 1. Resource Utilization Metrics

A time series of measurements for each of these metrics is gathered as the benchmark set runs. Each microbenchmark is executed for 10 min. Table 2 shows a fragment of a collected resource usage trace of the executed benchmarking set.

The time and workload type of each measurement are stored to simplify data processing and allow for the possibility of targeted benchmark reruns.

We monitor three CPU related metrics since different types of activities may have different virtualization overheads. For example, user space processing such as simple arithmetic operations performed by an application are unlikely to have much overhead

Time Interval	Benchmark ID	User CPU % M_1	Kernel CPU % M_2	IO.Wait CPU % M_3	Received Packs/s M_4	Sent Packs/s M_5	Received Bytes/s M_6	...	Read Blocks/s M_{10}	Write Blocks/s M_{11}	CPU VM %	CPU Dom0 %
1	1	22	8	0.5	500	335	210000	...	80	120	24	8
2	1
...

Table 2. A Fragment of a Collected Resource Usage Profile

in current virtualization platforms. In contrast, tasks which occur in kernel space, such as context switches, memory management, and I/O processing, are likely to have a higher level of overhead since they can require traps to the hypervisor.

We measure both the packet rates and byte rates of the network interfaces since different platforms may handle I/O virtualization in different ways. For example, prior to Xen version 3.0.3, incoming network packets were passed between Dom-0 and the guest domain by flipping ownership of memory pages, thus the overhead associated with receiving each packet was independent of its size [11]. Newer versions of Xen directly copy packets from Dom-0 to the guest domain rather than using page flipping, thus the overhead is also related to the number of bytes received per second, not just the number of packets. We differentiate between sending and receiving since these paths may have different optimizations.

We split disk measurements into four categories based on similar reasoning.

Table 2 shows a fragment of a collected resource usage trace of the executed benchmarking set. The last two columns in the profile report CPU utilization of the guest VM and Dom-0 in the virtualized environment, while the previous columns indicate the measured values for the same time interval and benchmark when executed on the native system.

4 Model Generation

This section describes how to create models which characterize the relationship between a set of resource utilization metrics gathered from an application running natively on real hardware and the CPU requirements of the application if it were run on a virtual platform. Two models are created: one which predicts the CPU requirement of the virtual machine running the application, and one which predicts the Dom_0 CPU requirements when it performs I/O processing on behalf of the guest domain.

The model creation employs the following *three key components*:

- A *robust linear regression* algorithm that is used to lessen the impact of outlier data points.
- A *stepwise regression* approach that is employed to include only the most statistically significant metrics in the final model.
- A *model refinement* algorithm that is used for post-processing the training data to eliminate or rerun erroneous benchmarks and to rebuild a more accurate, final model.

4.1 Model Creation

To find the relationship between the application resource usage in native and virtualized systems we use the resource usage profile gathered from a set of microbenchmarks run in both the virtual and native platforms of interest (see Table 2.)

Using values from the collected profile, we form a set of equations which calculate the Dom-0 CPU utilization as a linear combination of the different metrics:

$$\begin{aligned}
 U_{dom0}^1 &= c_0 + c_1 * M_1^1 + c_2 * M_2^1 + \dots + c_{11} * M_{11}^1 \\
 U_{dom0}^2 &= c_0 + c_1 * M_1^2 + c_2 * M_2^2 + \dots + c_{11} * M_{11}^2 \\
 &\dots \qquad \qquad \qquad \dots
 \end{aligned} \tag{1}$$

where

- M_i^j is a value of metric M_i collected during the time interval j for a benchmark executed in the native environment;
- U_{dom0}^j is a measured CPU utilization for a benchmark executed in virtualized environment with the corresponding time interval j .

Let $c_0^{dom0}, c_1^{dom0}, \dots, c_{11}^{dom0}$ denote the approximated solution for the equation set (1). Then, an approximated utilization \hat{U}_{dom0}^j can be calculated as

$$\hat{U}_{dom0}^j = c_0^{dom0} + \sum_{i=1}^{11} M_i^j \cdot c_i^{dom0} \tag{2}$$

To solve for c_i^{dom0} ($0 \leq i \leq 11$), one can choose a regression method from a variety of known methods in the literature. A popular method for solving such a set of equations is Least Squares Regression that minimizes the error:

$$e = \sqrt{\sum_j (\hat{U}_{dom0}^j - U_{dom0}^j)^2}$$

The set of coefficients $c_0^{dom0}, c_1^{dom0}, \dots, c_n^{dom0}$ is the model that describes the relationship between the application resource usage in the native system and application CPU usage in Dom-0.

We form a set of equations similar to Eq. 1 which characterize the CPU utilization of the virtual machine by replacing U_{dom0}^i with U_{vm}^i . The solution $c_0^{vm}, c_1^{vm}, \dots, c_n^{vm}$ defines the model that relates the application resource usage in the native system and application CPU usage in the virtual machine running the application.

To deal with outliers and erroneous benchmark executions in collected data and to improve the overall model accuracy, we apply a more advanced variant of this regression technique described in the next sections.

Robust Linear Regression Our training data is gathered by an automated benchmark system which must run identical workloads on both the native hardware system and the virtual platform. Slight benchmark timing errors or anomalous background processes can skew the measurements, leading to incorrect data points. With ordinary least squares

regression, even a few bad outliers can significantly impact the model accuracy, because it is based on minimizing the overall absolute error across multiple equations in the set.

To decrease the impact of occasional bad measurements, we employ iteratively reweighted least squares [12]. This technique is from the Robust Regression family of methods designed to lessen the impact of outliers.

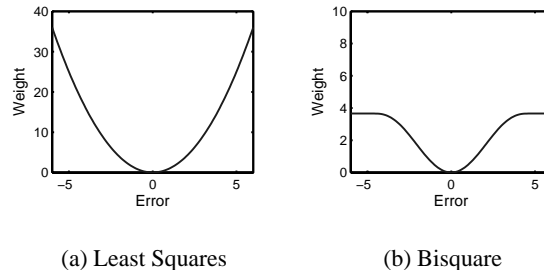


Fig. 3. Comparison of regression weighting functions.

Ordinary least squares regression works by creating an initial linear fit for a set of data points and then refining that fit based on the error between each data point and the line. Points pull the line towards them based on their weight, which is equal to the square of the distance from the data point to the line. Thus the weight of a point increases quadratically with its error as shown in Figure 3(a). This can lead to a few bad data points having more sway over the regression line than they should. In contrast, our robust regression technique uses a bisquare weighting function which is initially quadratic, but then levels off, lessening the weight of data points with high error as illustrated in Figure 3(b).

Stepwise Regression A direct (naive) linear regression approach would attempt to set non-zero values to all the model coefficients in order to produce the minimal error when the model is applied to the training set. However, this may lead to poor prediction accuracy when the model is later applied to other data sets, as the model may have become too finely tuned to the training set alone. In statistical terms, the model may “overfit” the data if it sets values to some coefficients to minimize the random noise in the training data rather than to correlate with the actual CPU utilization. In order to create a model which utilizes only the statistically significant metrics, we use stepwise linear regression to determine which set of input metrics are the best predictors for the output variable [8].

The algorithm initializes with an “empty” model which includes none of the eleven possible metrics. At each following iteration, a new metric is considered for inclusion in the model. The best metric is chosen by adding the metric which results in the lowest mean squared error when it is included. Before the new metric is included in the model, it must pass an F-test which determines if including the extra metric results in a statistically significant improvement in the model’s accuracy. If the F-test fails, then the algorithm terminates since including any further metrics cannot provide a significant benefit. The coefficients, for the selected metrics are calculated using the robust

regression technique described previously. The coefficient for each metric not included in the model is set to zero. We have found this procedure to be effective at selecting a subset of the metrics for an accurate model without requiring more advanced statistical techniques [1]. Stepwise regression is useful to identify the subset of essential metrics in order to reduce the amount of collected data for application usage profiles. While we could have limited traces to only gathering the metrics particularly important for Xen, we choose to initially gather a larger set of metrics and then pare it down using stepwise regression since different virtualization platforms may require different metrics.

Model Refinement Our use of robust linear regression techniques helps lessen the impact of occasional bad data points, but it may not be effective if all measurements within a microbenchmark are corrupt (this can happen due to unexpected background processes on the server, timing errors at the client, or network issues). The correct execution phase of the microbenchmark suite is a prerequisite for building an accurate model between native and virtualized platforms. If some microbenchmarks have failed or collected data were corrupted then it can inevitably impact the model outcome.

In order to automate the model generation process and eliminate the need for manual analysis of these bad data points, we must automatically detect erroneous microbenchmarks and either rerun them or remove their data points from the training set. At runtime, it can be very difficult to determine whether a benchmark is executed correctly, since the resource utilization cannot be known ahead of time, particularly on the virtual platform which may have unpredictable overheads. Instead, we wait until all benchmarks have been run and an initial model has been created to post process the training set and determine if some benchmarks have anomalous behavior.

First, we compute the mean squared error (MSE) for all data points (i.e., all microbenchmarks): let us call it e_{mean} , as well as the standard deviation of the squared errors: let us call it e_{std} . Then the model created from the full benchmark set is applied back to each microbenchmark i individually to calculate the mean squared error for that benchmark: let us call it e_i . Microbenchmarks with high error values can then be easily separated so that they can either be rerun or removed from the training set. If for microbenchmark i the following conditions is valid:

$$e_i > e_{mean} + 2 \times e_{std}$$

then microbenchmark i is considered to have abnormally high error, and it is eliminated from the set. Each microbenchmark contains approximately 20 data points. Assuming the errors of the microbenchmarks are independent and identically distributed, it follows from the central limit theorem that the probability of the mean of the squared errors from a microbenchmark of 20 data points to fall over two standard deviations (e_{std}) away from the benchmark mean (e_{mean}) is approximately only 10^{-18} . We note, however, that this result is only approximate since anomalous data points affect the mean (e_{mean}) and standard deviation computations (e_{std}).

4.2 Model Application

Once a model has been created, it can then be applied to resource utilization traces of other applications in order to predict what their CPU requirements would be if trans-

ferred to the virtual environment. Resource usage traces of the application are obtained by monitoring the application in its native environment over time. The traces must contain the same resource metrics as presented in Table 2, except that CPU utilizations of VM and Dom-0 are unknown and need to be predicted. Applying the model coefficients $c_0^{dom0}, c_1^{dom0}, \dots, c_{11}^{dom0}$ and $c_0^{vm}, c_1^{vm}, \dots, c_n^{vm}$ to the application usage traces in native environment (using Equation 1), we obtain two new CPU usage traces that estimate the application CPU requirements in Dom-0 and the virtual machine.

4.3 Model Scaling

Within similar hardware platforms, we would like to reuse models so that we do not need to repeatedly run the microbenchmark set on every pair of systems in a data center. For native systems within the same processor family, a single model can be created, and then traces from other systems can be “scaled to match” the model’s expected input units.

The microbenchmark suite is run on one of the native systems and on the destination virtual system to create an initial model. To apply this model to an application running on a different native system, the CPU coefficients in the model must be scaled based on the difference between the CPU speeds of each system. For example, if the initial model was created on a native system running at 2.6GHz, it can be scaled to be used for a 1GHz native system by multiplying each of the CPU coefficients included in the model by $\frac{1.0}{2.6} = 0.625$. This is equivalent to multiplying the CPU measurements from the 1.0Ghz trace by the same ratio, effectively scaling them down to what the CPU utilization would have been if the system was running at 2.6GHz. Only the CPU based coefficients need to be scaled since the I/O based metrics should be identical regardless of the CPU speed.

This technique assumes a linear relationship between CPU utilization of processors at different speeds.⁵ While such a scaling technique provides a conservative estimate of required CPU resources for an application that is executed at processors with higher speeds within the same family and with similar cache layouts, it is not the case for processors from different architectures.

5 Experimental Evaluation

In this section, we first try to justify a set of our choices presented in earlier Sections 3 and 4: why these *metrics*? why these *microbenchmarks*? why this *model creation process*? After that, we evaluate the effectiveness of our models under several realistic web application workloads on two different hardware platforms.

⁵ Clearly, we do not assume a linear relationship in application performance on the processors at different speeds. This technique allows a pessimistic, conservative estimate of the application CPU demands on the processors with higher speeds. In reality, the application CPU demands might be lower (due to memory accesses and I/O operations).

5.1 Implementation Details

Our implementation and evaluation has centered on the Xen virtualization platform. In our evaluation, both the native systems and virtual machines run the Red Hat Enterprise Linux 5 operating system with Linux kernel 2.6.18-8. We use Xen version 3.0.3-rc5.

Monitoring resource utilization in the native environment is done with the `sysstat` package [32] commonly used in Linux environments. The virtual CPU utilizations are measured using `xentop` and `xenmon`, standard resource monitoring tools included with the `xen` distribution. Statistics are gathered for 30 second monitoring windows in both environments. We have experimented with both finer grain and longer intervals and found similar results. The system is configured in such a way that Dom-0 resides on a separate CPU.

We evaluate our approach using two realistic web applications:

- *RUBiS* [3] is an auction site prototype modeled after eBay.com. A client workload generator emulates the behavior of users browsing and bidding on items. We use the Apache/PHP implementation of RUBiS version 1.4.3 with a MySQL database.
- *TPC-W* [34] represents an e-commerce website (modeled after Amazon.com) implemented with Java servlets running on Tomcat with a MySQL backend database.

Both applications have an application and a database tier. We profile and predict the resource requirements of the application server tier; the databases are hosted on a separate server which is sufficiently provisioned so that it will not become a bottleneck.

We have tested our approach on two different hardware platforms with the following details:

- HP ProLiant DL385, 2 processors: AMD Opteron model 252 2.6GHz with 1MB L2 single-core, 64-bit; 2 x 2GB memory; 2 x 1 Gbit/s NICs, 72 GB 15K U320 Disk.
- HP ProLiant DL580 G2, 4 processors: Intel Xeon 1.6 GHz with 1MB L2 processor, 32-bit; 3 x 2GB memory; 2 x 1 Gbit/s NICs, 72 GB 15K U320 Disk.

5.2 Importance of Modeling I/O

Our system generates models based on up to eleven different resource utilization metrics, here we evaluate whether such complexity is warranted, or if a simple model based solely on scaling CPU requirements is a viable approach. In the simplified approach, a model is created using the same model generation techniques as described in Section 4, except that instead of using all eleven metrics, only a single Total CPU metric is used to predict the virtual machine CPU needs. We produce a model using each technique to predict the CPU requirements of the guest domain, since, intuitively, it is more likely that the simplified model will perform better when predicting VM CPU needs than when predicting Dom-0 since the latter is scheduled almost exclusively for handling I/O.

Since our models are created with stepwise regression, not all of the eleven possible metrics are included in the final model. The Dom-0 model uses five metrics: Kernel CPU, I/O Wait, Rx Packets/sec, Tx Packets/sec, and Disk Write Req/sec. Dom-0's CPU utilization is dominated by I/O costs, so a large number of I/O related metrics

are important for an accurate model. In contrast the virtual machine model uses only three metrics: User Space CPU, Kernel CPU, and RX Packets. We compare this multi-resource VM model to the CPU-Scaling based model which uses only the Total CPU metric (equal to the sum of User Space and Kernel CPU).

We evaluate the performance of these two models by training them on our microbenchmark set and then comparing the error when the models are applied back to the training data. Figure 4 shows the error CDF for each model, showing the probability that our predictions were within a certain degree of accuracy for the virtual machine.

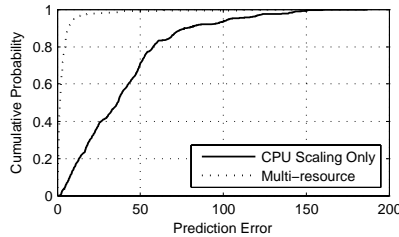


Fig. 4. Using CPU as the only prediction metric leads to high error.

Our multiple resource model performs significantly better than the CPU scaling approach; the 90th error percentile using our approach is 5% while the scaling approach is 65%. Without information about I/O activities, the simple model cannot effectively distinguish between the different types of benchmarks, each of which has different levels of overhead. Even though the VM model only includes one I/O metric, splitting CPU into User and Kernel time acts as a surrogate for detecting high levels of I/O. Our results suggest that I/O activity can cause significant changes in the CPU requirements of both Dom-0 and the guest domain: Dom-0 since it must process the I/O requests, and the guest because of the increased number of hypercalls required for I/O intensive applications.

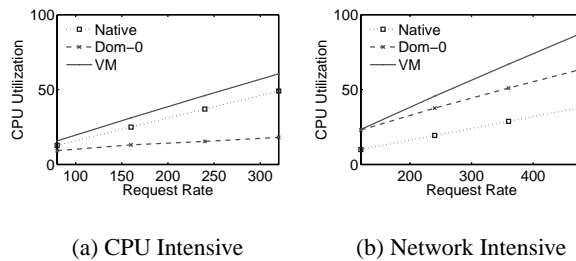


Fig. 5. I/O intensive applications exhibit higher virtualization overheads.

Figure 5 presents profiles of some of our CPU and network intensive microbenchmarks. The CPU intensive application exhibits only a small virtualization overhead oc-

curing for the VM CPU requirements and Dom-0 also has relatively low CPU needs. In contrast, the network intensive application has a significantly higher requirement in Dom-0 as well as a much larger increase in VM CPU requirements relative to the native CPU utilization. This further demonstrates why creating a model using only the native CPU metric is incapable of capturing the differences in overhead caused by I/O requests.

5.3 Benchmark Coverage

In this experiment we examine how the three different benchmark types each add useful information and examine the training set error of our model. Table 3 illustrates how using only a single type of microbenchmark to build a model can produce very high error rates when applied to applications with different workload characteristics.

		Test Set Median Error %		
		CPU	Net	Disk
Training Set	CPU	0.36	670	13
	Net	11	3.4	16
	Disk	7.1	1798	1.2
	All	0.66	1.1	2.1

Table 3. Using a subset of benchmarks leads to poor accuracy model when applied to data sets with different type of I/O.

For example, training the model solely with the CPU intensive microbenchmarks provides accuracy within 1% when applied back to the same kind of CPU intensive workloads, but the median error rises to 670% when applied to the network intensive data. This happens because the CPU benchmark includes only very low network rates. When a model based solely on that data tries to predict the CPU needs of the network intensive applications, it must extrapolate well beyond the range of data it was trained with, resulting in wildly inaccurate numbers. The bottom row in the table corresponds to using *all* of the benchmark data to create a model. This provides a high degree of accuracy in all cases – while a specialized model may provide higher accuracy on data sets very similar to it, we seek to build a general model which will be effective on workloads with a range of characteristics.

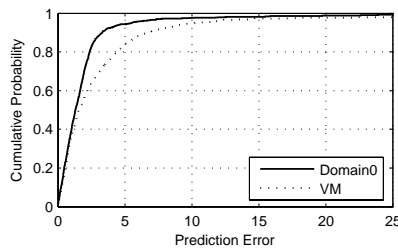


Fig. 6. CDF error of the training set on the Intel 4 -CPU machine.

Figure 6 shows the error CDF when *all* of our benchmark data is used to create a model and then the model is validated by applying back to the training set. The error is quite low, with 90% of the predictions being within 3% for Dom-0 and 7% for the virtual machine. This confirms our hypothesis that a single linear model can effectively model the full range of training data.

5.4 Benchmark Error Detection

Our profiling system runs a series of microbenchmarks with identical workloads on both the native and virtual platforms. This experiment tests our anomalous benchmark detection algorithm. To be effective, it should be able to detect which benchmarks did not run correctly so that they can be either rerun or eliminated from the training set. If the detection scheme is too rigorous, it may eliminate too many data points, reducing the effectiveness of the model.

We first gather a set of training data where 10 percent of the benchmarks are corrupted with additional background processes. Figure 7 shows the change in model accuracy after the error detection algorithm eliminates the malfunctioning microbenchmarks. We then gather a second training set with no failed benchmarks and run the error detection algorithm on this clean data set. We find that the model performance before and after the error detection algorithm is identical since very few data points are eliminated.

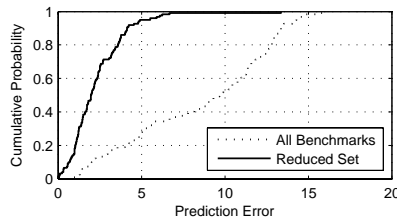


Fig. 7. Automatic benchmark elimination can increase model accuracy.

While it is possible for these errors to be manually detected and corrected, our goal is to automate the model creation procedure as much as possible. The error detection algorithm reduces the human interaction required to train and generate high quality models.

5.5 Model Accuracy

To test the accuracy of a model, we use it to predict the CPU requirements of a test application based on a trace of the application running natively. We then run the test application within the virtual environment to determine the prediction error. In this section we evaluate our models on both the RUBiS and TPC-W web applications. These experiments were run on the Intel system described previously.

We create a variable rate workload for RUBiS by incrementally spawning clients over a thirty minute period. The system is loaded by between 150 and 700 simultaneous

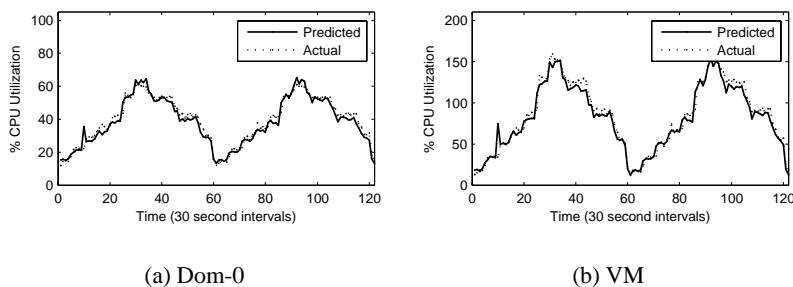


Fig. 8. Prediction accuracy of the RUBiS web application.

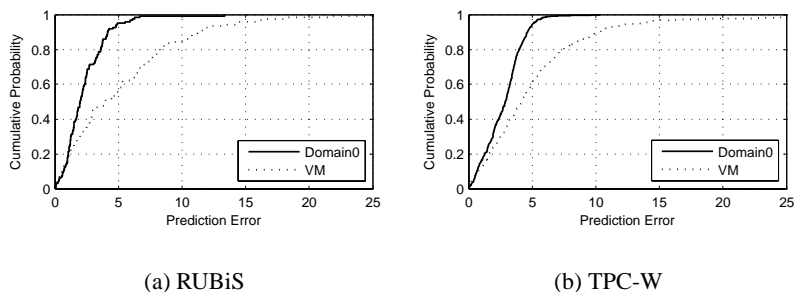


Fig. 9. Error rates on the Intel platform.

clients. This workload is repeated twice to evaluate the amount of random variation between experiments. We record measurements and make predictions for 30 second intervals. Figure 8 compares the actual CPU utilization of the RUBiS application to the amount predicted by the model. Note that the virtual machine running RUBiS is allocated two virtual CPUs, so the percent utilization is out of 200.

Figure 9(a) shows a CDF of the models' prediction error. We find that 90% of our predictions for Dom-0 are within 4% accuracy, and within 11% for predicting the virtual machine's CPU utilization. Some of this error is due to model inaccuracy, but it can also be due to irregularities in the data used as input to the model. For example, there is a spike in the predicted CPU requirements of both Dom-0 and the VM around time interval 10. This spike was caused by a background process running for a short period when RUBiS was run in the native environment. Since the predicted values are based on these native measurements, they mistakenly predict the virtual CPU requirements to spike in the same way.

We have also validated our model on the TPC-W application. We create a changing workload by adjusting the number of emulated clients from 250 to 1100 in a random (but repeatable) pattern. Figure 9(b) presents the error distribution for TPC-W. The error for this application is almost identical to RUBiS, with 90th percentile error rates of 5% and 10% for Dom-0 and the virtual machine respectively.

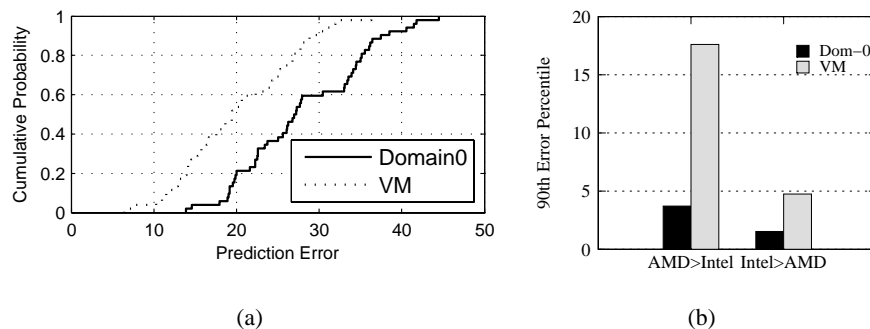


Fig. 10. (a) Using a single model for different architectures is ineffective, (b) but cross platform models are feasible.

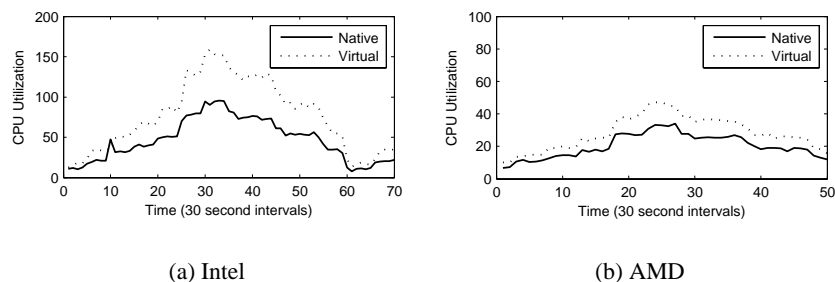


Fig. 11. Comparison of CPU overhead on different hardware platforms.

5.6 Cross Platform Modeling

In many server consolidation scenarios, the transition from a native to a virtual platform is accompanied by a change in the underlying hardware. However, using a single model for multiple hardware platforms may be ineffective if they have different overhead costs. Attempting to apply the model for the Intel system to the AMD system results in high error rates as shown in Figure 10(a). To investigate why these two platforms exhibit such a large difference, we compare the CPU required by the RUBiS application in the native and virtual environments on both platforms in Figure 11. Not including the Dom-0 requirements, the Intel system requires approximately 1.7 times as much CPU in the virtual case as it does natively. On the AMD system, the increase is only about 1.4 times. The different scaling between the native and virtual traces in each platform suggest that a single model cannot be used for both platforms.

We test our modeling approach’s ability to determine the relationship between native and virtual systems running on different hardware platforms by executing an identical set of microbenchmarks on the Intel and AMD platforms in both the native and virtual environments. Using this data, we create two models, one which relates a native usage profile of the Intel platform to a virtual usage profile of the AMD system and one which relates the native AMD system to the virtualized Intel system.

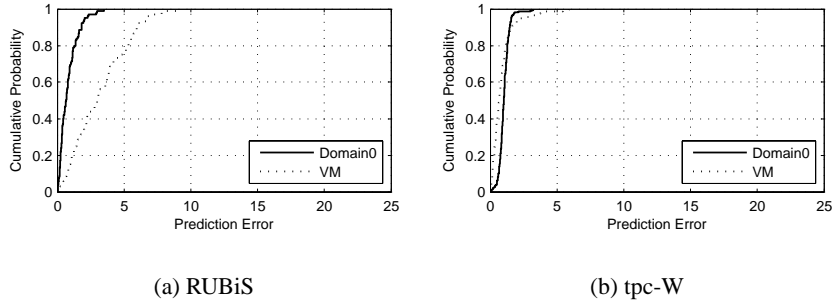


Fig. 12. Model accuracy on the AMD platform with scaled CPU speeds.

Figure 10(b) presents the 90th error percentiles when these cross platform models are used to predict the CPU needs of both the TPC-W and RUBiS workloads. The cross platform models are very effective at predicting Dom-0 CPU needs, however the VM prediction error is higher, particularly for the AMD to Intel model. We propose two factors which may cause this jump in error. First, the AMD system has a significantly faster CPU than the Intel system, so translating the CPU component from one platform to the other requires a significant scale up factor. As a result, small variations in the CPU needs of the AMD system can result in larger fluctuations in the predicted CPU for the Intel system, leading to higher absolute error values. Secondly, cross platform models for predicting virtual machine CPU are typically more difficult than Dom-0 models. This is because Dom-0 models are predominantly based on I/O metrics such as packet reception rates and disk operations, which have similar costs on both platforms. In contrast, the VM model is primarily based on the CPU related metrics which may not have a linear relationship between the two platforms due to differences in the processor and cache architectures. However, it should be noted that in many cases, the AMD to Intel model performs better than the 90th error percentile indicates; the median error is only 5%, and all of the points with high error occur at the peaks of the RUBiS workload where the virtual CPU consumption exceeds 160%.

5.7 Model Scaling within CPU Families

Within similar hardware platforms, we would like to reuse models so that we do not need to repeatedly run the microbenchmark set between the different pairs of systems within the same processor family. In this experiment, we take advantage of the CPU frequency scaling capabilities of our AMD system to examine *i*) whether we can create an accurate model between the systems with different processor frequencies and *ii*) how sensitive the models are to changes in CPU speed.

By adjusting the Linux boot options and the CPU power management states, we setup our native AMD platform to run at 1.8GHz and restrict it to using only a single CPU. For the virtual platform, we utilize both CPUs (one assigned to the virtual machine and one for Dom-0) and set the clockrate at the full 2.6GHz. The microbenchmark

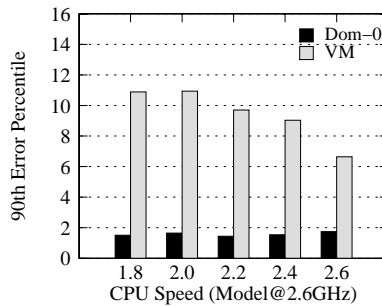


Fig. 13. A model can be scaled to apply to similar hardware platforms.

workloads are configured so that CPU utilization on the native system will not become saturated despite its lowered clock rate. To evaluate model accuracy, we use the RUBiS and TPC-W workloads described previously. Figure 12 presents the prediction error rates for each application; we find that the model is even more accurate than our initial tests on the Intel system, with 90th error percentiles of 3% and 6% for Dom-0 and the virtual machine respectively.

We next evaluate whether new models are required for each possible pair of CPU frequencies or if a single model can be scaled to work across a range of processor speeds.

An initial model is created relating the 2.6GHz native and virtual platforms (let us call it 2.6_native-to-2.6_virtual model) Then, given a collection of RUBiS usage traces in native AMD systems with different CPU speed ranging from 1.8 to 2.4GHz we aim to predict the RUBiS CPU requirements on the virtualized AMD system with a 2.6GHz processor.

First, the RUBiS CPU traces are scaled to match the model’s 2.6GHz original input, i.e., for the trace collected at 1.8GHz we use a scaling factor of $\frac{1.8}{2.6} = 0.69$ to project its CPU usage onto the 2.6GHz processor. Then the 2.6_native-to-2.6_virtual model is applied to predict RUBiS CPU requirements on the virtualized 2.6GHz AMD processor.

Figure 13 illustrates the 90th error percentile for each scenario. The error rate for the virtual machine starts at 7% for the 2.6_native-to-2.6_virtual model and increases to about 11% when applied to a trace running at 1.8GHz. The accuracy of predictions for Dom-0 is less influenced by the scaling factor since the CPU metrics play only a minor role in the model’s parameters.

Our modeling approach is insensitive to differences in the relative CPU speeds of the native and virtual platforms. Within similar processor families, a model can be scaled so that it can be applied to a range of CPU speeds while still maintaining reasonable accuracy, this reduces the number of models which must be created when transitioning between multiple hardware platforms.

6 Discussion

In this section, we discuss the impact of the application behavior on the accuracy of the prediction results and difficulties in understanding application performance, as well as challenges introduced by dynamic frequency scaling.

- *Impact of application behavior on resource use.*

The application behavior in the native system is represented via different resource usage traces over time. We use them to project the application CPU requirements in a virtual system. The intuition is that for different portions of these *native* traces there is a different scaling factor (resulting in CPU overhead) defined by system activities and operations performed by the application. By applying regression models to native platform application traces, we produce estimated CPU usage traces for for the same workload in the VM and Dom-0.

However, the timing for the application’s operations in the native and virtualized environments may be slightly different if the application has a strong “feedback loop” behavior.

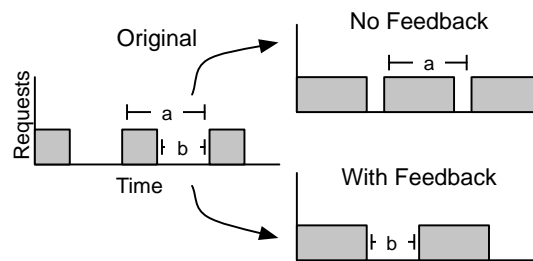


Fig. 14. Resource requirements in different environments is influenced by the amount of feedback in an application’s workload.

Figure14 illustrates the difference between an application with and without feedback. In the original application trace, a series of requests arrive, with their processing time indicated by the width of the rectangles. The value of a represents the time from the start of one request until the start of the next, while b is the time from the end of one request to the start of the next. When the same application is run on a different platform, the time to process a request may increase due to virtualization overhead. The two figures on the right represent how the trace would appear if the application does or does not exhibit feedback. Without a feedback loop, the time between the start of each request will remain a , even if the request processing time increases. This would occur if the requests are being submitted by a client on another machine sending at a regular rate. For an application with feedback, requests are processed then a constant delay, b , occurs before the next request is processed. The figure illustrates that when request processing times increase, applications with feedback may process fewer requests in a given time interval (due to a slowdown), i.e., its CPU overhead is “spread” across a longer time period, resulting in lower average CPU utilization.

It is impossible to tell if an application's workload has a feedback loop just by looking at resource utilization traces of the original application. So the estimated resource utilization produced by our model for the application with a "feedback loop" might be higher than in reality since such an application might consume CPU resources in virtualized environment "slower" than in native one due to the increased latency on the application's critical path.

- *Understanding Application Performance.*

While our models can accurately predict the changes in resource requirements for a virtualized application, they cannot directly model how application performance (ie. response time) will change. Unfortunately, this is a difficult challenge, akin to making performance predictions under different hardware platforms. Our approach tells system administrators the minimum amount of resources which must be allocated to a VM in order to prevent significantly reduced performance due to resource starvation. The application may still see some performance penalty due to the longer code path as requests go through the virtualization layer. To accurately predict this performance change would necessitate carefully tailored, application specific models.

Our approach helps in estimating the resource requirements that are necessary for the initial application placement in a virtualized environment. After the initial workload placement, specialized workload management tools may be used [15, 13] to dynamically adjust system resources to support the required application performance.

- *Challenges caused by dynamic frequency scaling.*

The latest server processors from Intel and AMD have power state hardware registers that enable control of performance and power consumption of the processor. These capabilities are implemented through Intel's Enhanced SpeedStep Technology and demand-based switching and through AMD's PowerNow with Optimized Power Management (OPM). With the appropriate ROM firmware or operating system interface, programmers can use the exposed hardware registers to dynamically modify the frequency and voltage of each processor based on the processor workload. Thus the processor operates in a high power state (at a maximum processor frequency) only when needed, thus reducing the overall system power usage. However, current OS-based monitoring tools report CPU utilization without taking into account the processor frequency information. Therefore, collected CPU utilization measurements might incorrectly represent application resource requirements when they are obtained from the system with a dynamic frequency scaling policy. In order to get correct (actual) usage traces, one needs to be aware of the power saving policy employed by the system and scale/normalize the CPU utilization traces by dynamically changing processor frequency.

7 Related Work

Virtualization Overheads: Virtualization is gaining popularity in enterprise environments as a software-based solution for building shared hardware infrastructures. VMware and IBM have released their benchmarks (VMmark and Grand Slam) for quantifying the performance of virtualized environments. These benchmarks aim to provide some basis for comparison of different hardware and virtualization platforms

in server consolidation exercises. However, they both are lacking the ability to characterize virtualization overhead compared to a native platform.

Application performance and resource consumption in virtualized environments can be quite different from its performance and usage profile on native hardware because of additional virtualization overheads (typically caused by I/O processing) and interactions with the underlying virtual machine monitor (VMM). Several earlier papers which describe various VMM implementations include performance results that measure the impact of virtualization overhead on microbenchmark or macrobenchmark performance (e.g., [5, 20, 38, 31, 2, 40, 18, 31, 7, 25]). The reported virtualization overhead greatly depends on the hardware platform that is used in such experiments. For example, previously published papers [5, 9] evaluating Xen’s performance have used networking benchmarks in systems with limited network bandwidth and high CPU capacity. However, there are cases where throughput degrades because CPU processing is the bottleneck instead of the network [22, 11]. In many virtualization platforms, the “amount” of CPU overhead is directly proportional to the “amount” of performed I/O processing [7, 11]. For example, it has been shown that networking packet rates are highly correlated with the measured CPU overhead [11]. Recent work attempts to reduce the performance penalty of network I/O by bypassing parts of the virtualization layer [19, 39] or optimizing it [26]. However, since these optimizations typically target only one source of virtualization overhead (network I/O), our modeling system can still be employed to provide useful information about the level of overhead incurred by a wider range of activities.

This extensive body of previous work has motivated us to select a set of microbenchmarks that “probe” system resource usage at different I/O traffic rates (both networking and disk) and then employ these usage profiles for predicting variable CPU overhead of virtualized environments.

Trace-based Approaches: In our work, we chose to represent application behavior via resource usage traces. Many research groups have used a similar approach to characterize application behavior and applied trace-based methods to support what-if analysis in the assignment of workloads to consolidated servers [35, 27, 29, 10]. There are a few commercial tools [14, 33, 37, 16] that employ trace-based methods to support server consolidation exercises, load balancing, ongoing capacity planning, and simulating placement of application workloads to help IT administrators improve server utilization. Since many virtualization platforms introduce additional virtualization overhead, the trace-based capacity planning and management solutions provide a capability to scale the resource usage traces of original workloads by a specified CPU-multiplier. For some applications it might be a reasonable approach, however, in general, additional CPU overhead highly depends on system activities and operations performed by the application. Simplistic trace-scaling may result in significant modeling error and resource over-provisioning.

System Profiling: Finally, there is another body of work [21, 30, 6, 28] that is closely related to our thinking and the approach presented in the paper. This body of work goes back to 1995, when L. McVoy and C. Staelin have introduced the *lmbench* – a suite of operating system microbenchmarks that provides a set of portable programs for use in cross-platform comparisons. Each microbenchmark was purposely created to

capture some unique performance problem present in one or more important applications. Although such microbenchmarks can be useful in understanding the end-to-end behavior of a system, the results of these microbenchmarks provide little information to indicate how well a particular application will perform on a particular system. In [6, 28], the authors argue for an application-specific approach to benchmarking. The authors suggest a vector-based approach for characterizing an underlying system by a set of microbenchmarks (e.g., *lmbench*) that describe the behavior of the fundamental primitives of the system. The results of these microbenchmarks constitute the *system* vector. Then they suggest to construct an *application* vector that quantifies the way that the application makes use of the various primitives supported by the system. The product of these two vectors yields a relevant performance metric. There is a similar logic in our design: we use a set of microbenchmarks to characterize underlying system and virtualization solution. Then we apply the derived model (analogy to a *system* vector) to the application usage traces (analogy to the *application* vector) and use it for predicting the resource requirements of applications when they are transferred to a virtual environment.

8 Conclusions

Our work is motivated by the need for improved estimates of application resource requirements when they are consolidated to virtual environments. To this end, we designed an automated approach for profiling different types of virtualization overhead on a given platform and a regression-based model that maps the native system profile into a virtualized one. This model can then be used to accurately assess the required resources and make workload placement decisions in virtualized environments.

Although such a model is created using data from synthetic benchmarks, the result is a general model which can be applied to traces from any other application in order to predict what its resource requirements will be on the virtual platform. We profile each platform using open source tools that can be easily deployed and executed on a wide range of hardware platforms within traditional or next generation data centers. We envision that each system in a NGDC will be augmented with a model that reflects the relationship between the *native* and *virtualized* system usage profiles. To enhance the usage of such a model we provide a model scaling approach for systems within the same processor family.

Our evaluation has shown that our automated model generation procedure effectively characterizes the different virtualization overheads of two diverse hardware platforms and that the models have median prediction error of less than 5% for both RUBiS and TPC-W. In future work we plan to experiment with more diverse application types and different virtualization platforms. We are also interested in how these modeling techniques can be used to predict the aggregate resource requirements of virtual machines collocated on a single host and to determine when an application's resource requirements are likely to exceed the virtual system's capacity.

References

1. C. Agostinelli. Robust Stepwise Regression. In *Journal of Applied Statistics*, Volume 29, Number 6, 2002.
2. I. Ahmad, J. Anderson, A. Holler, R. Kambo, and V. Makhija. An Analysis of Disk Performance in VMware ESX Server Virtual Machines. *Proc. of the Sixth Workshop on Workload Characterization (WWC'03)*, October 2003.
3. C. Amza, E. Cecchet, A. Chanda, A. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel. Specification and implementation of dynamic Web site benchmarks. *Proc. of WWC-5: IEEE 5th Annual Workshop on Workload Characterization*, October 2002.
4. Apache JMeter. <http://jakarta.apache.org/jmeter/>.
5. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM SOSP*, 2003.
6. A. Brown, M. Seltzer. Operating System Benchmarking in the Wake of Lmbench Case Study of the Performance of NetBSD on the Intel Architecture. In *Proc. of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.
7. L. Cherkasova and R. Gardner. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. *Proc. of USENIX Annual Technical Conference*, Apr 2005.
8. N. R. Draper and H. Smith. *Applied Regression Analysis*. John Wiley & Sons, 1998.
9. K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. *Reconstructing I/O*. Technical report, 2004.
10. D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper: Capacity Management and Demand Prediction for Next Generation Data Centers. *Proc. of the International IEEE Conference on Web Services (ICWS'2007)*, Salt Lake City, Utah, USA, 2007.
11. D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat: Enforcing Performance Isolation Across Virtual Machines in Xen. *Proc. of the ACM/IFIP/USENIX 7th Intl Middleware Conf. (Middleware)*, Melbourne, Australia, 2006.
12. P. W. Holland and R. E. Welsch. Robust regression using iteratively reweighted least-squares. In *Communications in Statistics - Theory and Methods* 6.9. 06 Oct. 2007.
13. HP-UX Workload Manager. <http://www.hp.com/products1/unix/operating/wlm/>.
14. HP Integrity Essentials Capacity Advisor. <http://h71036.www7.hp.com/enterprise/cache/262379-0-0-0-121.html>
15. IBM Enterprise Workload Manager. <http://www.ibm.com/developerworks/autonomic/ewlm/>.
16. IBM Tivoli Performance Analyzer. <http://www.ibm.com/software/tivoli/products/performance-analyzer/>
17. IBM eServer i5 Virtualization Grand Slam Benchmark: Executive Summary. www.ibm.com/servers/uk/eserver/series/literature/
18. S. King, G. Dunlap, P. Chen. Operating system support for virtual machines. *Proc. of the USENIX Annual Technical Conference*, San Antonio, Texas, 2003. www.hpl.hp.com/news/2006/apr-jun/technology.html
19. J. Liu, W. Huang, B. Abali, D. Panda. High Performance VMM-Bypass I/O in Virtual Machines. *Proc of Usenix AT* 2006.
20. D. Magenheimer and T. Christian. vBlades: Optimized paravirtualization for the Itanium processor family. *Proc. of USENIX Virtual Machine Research and Technology Symposium*, May 2004.
21. L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. *Proc. of the 1996 Winter USENIX*, San Diego, CA, Jan. 1996.

22. A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. Proc. of the First ACM/USENIX Intl. Conf. on Virtual Execution Environments (VEE), June 2005.
23. D. Mosberger, T. Jin. Httperf—A Tool for Measuring Web Server Performance. Proc. of Workshop on Internet Server Performance, 1998.
24. Oprofile. <http://oprofile.sourceforge.net>
25. P. Padala, X. Zhu, Z. Wang, S. Singhal, K. Shin. Performance Evaluation of Virtualization Technologies for Server Consolidation. HP Labs Technical Report HPL-2007-59, 2007.
26. Jose Renato Santos, Yoshio Turner, and G. (John) Janakiraman, Ian Pratt: Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. Proc of Usenix 2008.
27. J. Rolia, L. Cherkasova, M. Arlitt, A. Andrzejak. A Capacity Management Service for Resource Pools. Proc. of the 5th Intl. Workshop on Software and Performance (WOSP), Palma, Spain, 2005.
28. M. Seltzer, D. Krinsky, K. Smith, X. Zhang. The Case for Application-Specific Benchmarking. Proc. of the 1999 Workshop on Hot Topics in Operating Systems (HotOS VII), Rio Rico, AZ, March, 1999.
29. S. Seltzsam, D. Gmach, S. Krompass, A. Kemper. AutoGlobe: An Automatic Administration Concept for Service-Oriented Database Applications. Proc. of the 22nd Intl. Conf. on Data Engineering (ICDE), 2006.
30. C. Staelin and L. McVoy. mhz: Anatomy of a microbenchmark. Proc. of the USENIX Annual Technical Conference, New Orleans, LA, June 1998.
31. J. Sugerman, G. Venkitachalam, B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. Proc. of the USENIX Annual Technical Conference, Boston, Massachusetts, 2001.
32. Sysstat-7.0.4. <http://perso.orange.fr/sebastien.godard/>
33. TeamQuest: <http://www.teamquest.com>
34. TPC-W Benchmark. <http://www.tpc.org>
35. B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. Proc. of Operating Systems Design and Implementation (OSDI), Dec 2002.
36. VMmark: A Scalable Benchmark for Virtualized Systems. www.vmware.com/pdf/vmmark_intro.pdf
37. VMware Capacity Planner. www.vmware.com/products/capacity_planner/
38. C. Waldspurger. Memory resource management in VMware ESX server. In Operating Systems Design and Implementation. Proc. of Operating Systems Design and Implementation (OSDI), Dec 2002.
39. J. Wang, K. Wright, and K. Gopalan, XenLoop : A Transparent High Performance Inter-VM Network Loopback, Proc. of International Symposium on High Performance Distributed Computing (HPDC), Boston, MA, June 2008.
40. A. Whitaker, M. Shaw, and S. Gribble. Scale and Performance in the Denali isolation kernel. Proc. of Operating Systems Design and Implementation (OSDI), Dec 2002.
41. XenSource: <http://www.xensource.com/>