An Analysis of Performance Interference Effects in Virtual Environments

Younggyun Koh[1*], Rob Knauerhase[2], Paul Brett[2], Mic Bowman[2], Zhihua Wen[3*], Calton Pu[1]

[1] College of Computing
Georgia Institute of Technology
{young, calton@cc.gatech.edu}

[2] Intel Corporation
knauer@jf.intel.com,
{paul.brett, mic.bowman}@intel.com

[3] EECS Department
Case Western Reserve University
{zxw20@eecs.cwru.edu}

ABSTRACT

Virtualization is an essential technology in modern datacenters. Despite advantages such as security isolation, fault isolation, and environment isolation, current virtualization techniques do not provide effective performance isolation between virtual machines (VMs). Specifically, hidden contention for physical resources impacts performance differently in different workload configurations, causing significant variance in observed system throughput. To this end, characterizing workloads that generate performance interference is important in order to maximize overall utility.

In this paper, we study the effects of performance interference by looking at system-level workload characteristics. In a physical host, we allocate two VMs, each of which runs a sample application chosen from a wide range of benchmark and real-world workloads. For each combination, we collect performance metrics and runtime characteristics using an instrumented Xen hypervisor. Through subsequent analysis of collected data, we identify clusters of applications that generate certain types of performance interference. Furthermore, we develop mathematical models to predict the performance of a new application from its workload characteristics. Our evaluation shows our techniques were able to predict performance with average error of approximately 5%.

1. INTRODUCTION

Virtualization technology [1][24][25] offers many advantages to datacenter administrators and end users. By running multiple virtual machines (VMs) in a shared physical machine, virtualization enables high utilization of hardware resources. Live migration and easy restart of VMs improve manageability of large datacenters. Meanwhile, virtual machine technology provides strong isolation among virtual domains. For example, security isolation prevents a malicious application from attacking applications or accessing data in other domains. Fault isolation prevents one misbehaving application from bringing down the whole system. Environment isolation allows multiple operating systems to run on the same machine, accommodating legacy applications and cutting-edge software, each with a separate set of configurations and parameters.

Despite such advantages, we have observed that modern virtual machine technologies do not provide effective performance isolation. While the hypervisor (*a.k.a.* the virtual machine monitor) slices resources and allocates shares to different VMs, the behavior of one VM can still affect the performance of another adversely due to the shared use of resources in the system. Furthermore, the isolation provided by virtualization limits the visibility of an application in a VM into the cause of performance anomalies that occur in a virtualized environment. Specifically, a user running the same virtual machine on the same hardware at different times will see wide disparity in performance based on the work performed by other VMs on that physical host. We use the term *performance interference* to describe this phenomenon.

The performance interference in virtual environments differs from one in a traditional operating system in a few important aspects. First, multiple VMs on a hypervisor contain several independent resource schedulers, each of which is attempting to manage shared resources without visibility of the others. Therefore, the system suffers from non-obvious interference that underlies the OS and is outside the control of an OS or the hypervisor. Second, the guest OSes and applications inside a VM cannot – by the nature of isolation – be fully informed about other work in another domain, and therefore cannot deterministically measure the effects of performance interference. Moreover, while an OS can easily access detailed information of applications it runs, the hypervisor has very low visibility[1] into both the guest OS and its applications. Because of this, many of the possible state-of-the-art optimization techniques are difficult for the hypervisor to implement. Finally, some hypervisors insert another layer of abstraction by offloading certain operations such as I/O operations to service VMs. This additional redirection created by the hypervisor layer particularly affects the performance of I/O-intensive applications; for example, since the Xen hypervisor forces all the I/O operations from guest OSes to pass through a special device driver domain, the context-switches into and out of a device driver domain

[*] Most of this work was done while Koh and Wen were interns at Intel Corporation in the summer of 2006.

[1] Indeed, this opacity is often regarded as a feature, allowing the hypervisor to be less complex and therefore easier to optimize and debug.

result in a dimension of performance interference unique to virtualization.

In this paper, we study performance interference among the selected applications in our experimental virtual environments. By analyzing the system-level characteristics collected from the virtual environments, we have observed that a significant degree of performance interference exists when we run certain types of applications on shared hardware at the same time. To understand the performance behaviors of applications under performance interference, we classify applications using different metrics. Furthermore, we develop mechanisms to predict the expected performance scores of the applications running different types of workloads. Our mechanisms were able to predict scores with average error of approximately 5%.

The paper is organized as follows. Section 2 presents our experimental setup. Section 3 shows initial performance results to motivate our problems. In section 4, we identify performance interference by characterizing workloads. Section 5 introduces and evaluates our performance prediction mechanisms. We discuss the usefulness of our work and related work in section 6 and section 7. We conclude with summary and future work in section 8.

2. EXPERIMENTAL SETUP

2.1. Virtual Resource Allocation Environments

For our study of performance interference, we developed an experimental virtual resource allocation (VRA) environment. In this environment, we can remotely create numerous virtual machines with different resource specifications in different physical hosts. In each VM, we run a wide range of different applications to gain actual data for our analysis. The applications we use for our experiments are chosen from real-world workloads, such as compression, the compilation of source code, and rendering frames, as well as from well-known benchmark suites, such as the SPEC2000 CPU benchmark [23] and lmbench benchmark suite [21].

In this study, two VMs are created in a physical host, using the Xen hypervisor [1][25]. Each VM domain ("dom1" and "dom2", respectively) runs one of our VRA applications. Notationally, the application running in dom1 is called the "foreground" application, and the one in dom2 is the "background" application. A foreground application F running against background application B is denoted as F@B. Because the running time of each application varies, we ensure the background applications stay active by restarting them if needed in dom2 while foreground applications complete. We run the applications in a way that every application runs both as foreground and background, so as to construct an *n* x *n* matrix, containing all the possible combinations of measurement results.

For a hypervisor scheduler, we used the Borrowed Virtual Time scheduler (BVT) [5], which is one of the mature Xen hypervisor schedulers currently available.

2.2. VRA Applications

We carefully choose our test applications to stress various aspects of the system and hardware.

Add_double is a sub-benchmark program from the AIM benchmark suite [17]. *Add_double* measures the performance of double precision adding operations. *Analyser* is a benchmark program from the Freebench benchmark suite [19]. *Analyzer*'s performance is limited by the memory subsystem. *Bzip2* and *gzip* are popular compression applications. They are also included in the SPEC CPU2000 benchmark suite [23]. We run *bzip2* and *gzip* for compressing text files with the `-best` option. *Ccrypt* is a popular open-source encryption and decryption tool, and we encrypt a variety of text files in our experiment. For compression and encryption applications, we use files sufficiently large so as not to fit in the file cache of the guest operating systems.

Cachebench [18] is an open-source benchmark program that determines the performance of a memory subsystem. We wrote a program (*cachebuster*) that invalidates the cache content quickly by striding memory address space by the cache line size of the underlying physical processor. *Cat* and *grep* are popular Linux tools that generate a lot of disk read requests; we use large text files as input for these experiments. We also use *cp* and *dd* to generate disk write activity, copying small files into a complicated directory structure. *Iozone* [20] is another disk I/O benchmark suite, providing varied read and write tests. *Bw_mem* is a memory benchmark program included in the lmbench benchmark suite [21].

We also measure the performance of building a source code package, specifically a recent version of the Apache web server [16]; this test is referred to simply as *make*. *Povray* [22] is a frame rendering tool for 3-D graphics. Finally, we wrote a very simple program named *spinlock* that loops infinitely, consuming almost nothing but CPU cycles.

Some of the VRA applications produce explicit performance scores. For the rest of the applications, we regard the inverse of the elapsed time of one run as its performance score. We summarize our applications in Table 2-1.

Table 2-1 VRA Test Applications

Name	Major resource used	Performance measurement
Add_double	CPU	Score
Analyser	Memory	Elapsed time
Bw_mem	Memory	Score
Bzip2	Mixed	Elapsed time
Cat	Disk	Elapsed time
Cachebench	Memory	Score
Cachebuster	Memory	Elapsed time
Ccrypt	Mixed	Elapsed time
Cp	Disk	Elapsed time
Dd	Disk	Elapsed time
Grep	Disk	Elapsed time
Gzip	Mixed	Elapsed time
Iozone	Disk	Score
Make	Mixed	Elapsed time
Povray	Mixed	Elapsed time
Spinlock	CPU	Elapsed time

2.3. Normalized Performance Score

Since we are interested in finding out the extent to which performance is affected by interference generated by an application running in the other domain, we use degradation from standard performance as a normalized score. To calculate normalized scores of an application, first we define idle performance scores as the scores of applications when they are running against an idle domain[2]. Then, we calculate a normalized score of application F running against B, dividing the score of F by its idle performance score. Thus, we have NS(F@B), a normalized score of F against B,

$$NS(F@B) = PerformScore(F@B) / PerformScore(F@Idle)$$

Since we are interested the overall performance of applications, we define NS (F+B), combined performance of two applications, F and B, in each domain,

$$NS(F+B) = NS(F@B) + NS(B@F)$$

NS(F@B) and NS(B@F) are measured in two separate experiments.

3. Performance Interference

We present combined performance results for sample applications in Figure 3-1. Note that the performance scores are normalized. Thus, any application running standalone has normalized performance of 1, as in *bzip2*. Ideally, the expected combined performance is 1, assuming each application uses exactly half resources in the system. The dashed line in the graph represents the ideal expected performance.

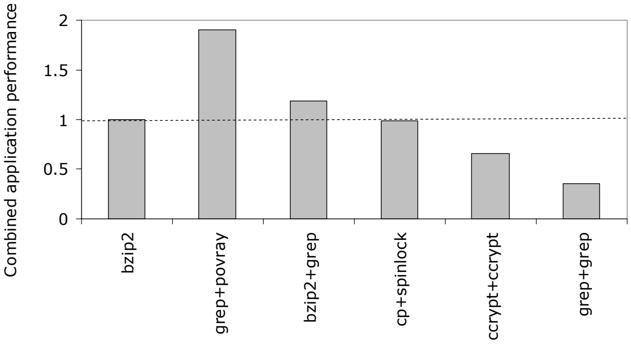

Figure 3-1 Varying performance with different combinations of applications

We notice, however, that combined performance varies substantially with different combinations of applications. Applications that rarely interfere with each other achieve performance close to the standalone performance, resulting in combined performance close to 2, as in *grep+povray*. However, some combinations interfere with each other in an adverse way, and their combined performance drops significantly – for example, down to 0.35, as in *grep+grep*.

The performance interference we observe here derives from state in the shared physical resources between VMs. For example, if an application runs in a VM by itself, it can use techniques to warm the cache(s) and take advantage of this faster memory for improved performance. However, when the hypervisor switches to another VM (e.g. at the end of each quantum), the new VM gains access to the cache(s). Consequently, when the original VM regains access to the cache, there is high probability that the cache is at least partially cold. Worse yet, because the hypervisor is opaque in its representation of physical hardware, the application and the guest OS are not aware of these changes, which makes it difficult to apply traditional optimization techniques.

Other resources that provide some amount of state include disk storage. For example, a streaming read will be interrupted by any disk access that another VM does. While techniques such as anticipatory scheduling in the hypervisor [9] helps reduce overhead in certain cases, this interference can have a significant effect on performance, depending on the access pattern of each VM, the time quantum of the hypervisor, and the nature of the storage system.

4. Analysis of Performance Interference

4.1. System-level Workload Characteristics

To capture VM behaviors that generate performance interference, we collect system-level workload characteristics through an instrumented hypervisor as described earlier. We decide to collect system-level characteristics for workload characterization in two reasons. First, these characteristics are independent of the underlying micro-architecture, so we can draw comparisons across physical hosts of different types. Second, by measuring characteristics from outside of the VM environment, we eliminate the needs of simulation or re-compilation of applications or modification of the guest OS.

In our experimental setup, we collected the following 10 different workload characteristics per each VM:

Average CPU utilization (`cpuutil`). We derive average CPU utilization by dividing cputime used for a VM by the elapsed (wall-clock) time of the VM. Cputime is collected by the hypervisor.

Cache hits and misses per second (`cachehits`, `cachemisses`). Cache behavior is an important metric to understand the memory usage of a VM. Our instrumented hypervisor provides us with cache hit and miss numbers from counters in the processor.

Virtual machine switches per second (`vmswitches`, `novmswitches`). We measure how many times the hypervisor switches control to a different VM (`vmswitches`) or returns to the same VM (`novmswitches`).

I/O blocks per second (`blocks`). We also measure how many times a VM is blocked due to I/O waiting, when it relinquishes the remainder of its quantum to the hypervisor.

Disk reads and writes issued per second (`reads_issued`, `writes_issued`). Disk reading and writing time per VM (`time_reading`, `time_writing`). The numbers of read and write requests issued to (virtual) disk device drivers and the time spent for the requests are good indicators of I/O.

[2] This allows us to eliminate the virtualization overhead, the optimization of which is a separate matter.

Table 4-1 System-level characteristics of benchmark applications against idle

	cpuutil	cachehits (K)	cachemisses (K)	vmswitches	novmswitches
add_double	1.00	310.31	8.74	63.87	58.66
analyser	1.00	42299.19	7821.22	61.00	59.97
bzip2	0.96	46008.36	3581.26	96.77	56.31
cachebench	1.00	144327.44	1683.38	55.94	62.20
cachebuster	1.01	17531.42	17242.03	71.70	59.12
cat	0.22	12790.63	502.35	619.39	2.08
ccrypt	0.96	13809.18	359.98	327.10	3.48
cp	0.84	59524.61	2343.92	511.30	47.36
dd	0.82	54976.78	2152.38	571.84	45.59
grep	0.02	1753.52	47.37	403.74	0.00
gzip	0.98	142100.17	228.14	176.39	28.16
iozone	0.11	8631.83	798.31	1607.07	0.59
bw_mem	1.01	10107.07	51040.85	81.93	70.71
make	0.98	92662.22	1567.48	94.36	59.86
povray	1.00	59365.30	5.88	58.48	60.65
spinlock	1.00	276.02	10.86	61.28	61.93
	reads_issued	time_reading	writes_issued	time_writing	blocks
add_double	0.67	2.66	3.68	21.62	2.31
analyser	1.29	2.57	1.03	0.43	1.78
bzip2	83.32	252.42	0.57	0.27	11.65
cachebench	0.22	1.60	0.34	0.16	0.58
cachebuster	1.07	4.11	0.74	0.00	6.66
cat	186.02	791.79	1.76	25.09	266.69
ccrypt	475.42	997.34	1.31	26.62	4.22
cp	15.02	37.49	711.79	8628.56	271.48
dd	15.92	37.87	750.01	9263.64	325.66
grep	292.65	978.97	3.23	57.51	401.60
gzip	201.77	314.82	0.83	0.65	1.61
iozone	631.20	2436.34	1192.50	16369.25	1581.08
bw_mem	1.46	9.75	2.44	0.00	14.87
make	10.10	32.55	24.64	83.32	16.09
povray	0.36	1.60	0.60	0.00	0.57
spinlock	0.24	1.19	0.54	0.00	3.33

In Table 4-1, we show workload characteristics of our benchmark applications running against an idle domain. Notice that CPU- and memory-intensive applications have high average CPU utilization, while I/O-intensive applications do not fully consume their assigned CPU quanta. Some real-world applications, such as *bzip2* and *make*, have high I/O indicators, since they read input files from disk. Memory-intensive benchmark programs show high cache numbers, particularly high miss numbers. Compression applications have relatively high cachehits but low cachemisses, which implies that they frequently access a small working set of memory (and are likely tuned for performance in this way).

4.2. Performance against Different Workloads

As described earlier, the performance score of each application varies depending on the type of application running in the other domain. We show our measured performance scores of benchmark applications in Figure 4-1.

Ideally, since we run two VMs in a physical host, the expected performance score of each application is 0.5, on the assumption that each application makes use of exactly half of resources in real hardware. However, we notice that performance scores of applications are greatly affected by background applications. *grep* has the largest variance for performance scores among the test applications, while *dd* has the least. Meanwhile, *iozone* as a background application gives the most diverse performance scores for foreground applications, *cp* as a background does the least. Note that the normalized performance of all the applications against an idle domain is 1.

4.2.1. Cache Interference

Cachebuster is a program of our own design that invalidates processor cache quickly, walking the address space by striding by the cache-line size of the underlying physical processor. Against *cachebuster*, some of the applications, particularly memory-intensive ones, such as *cachebench*, *cachebuster*, and *analyser*, suffer from significant performance interference.

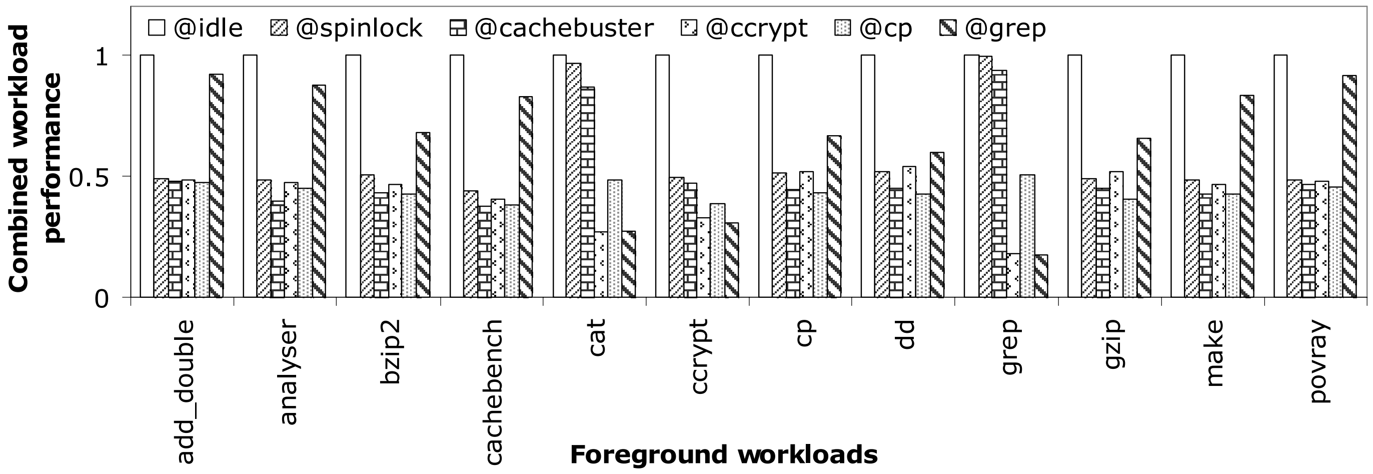

Figure 4-1 Performance score variations for selected foreground and background workloads

Table 4-2 shows workload characteristics of *analyser* and *cachebuster* when they are running against each other. Note that both programs use CPU quanta evenly. Because *analyser* is now scheduled for half of the time, one would naïvely expect to see *analyser* having half cache hit and miss numbers of the idle case, that is, 42299.19 / 2 = 21149.60 and 7821.22 / 2 = 3910.6. However, with *cachebuster* in dom2, *analyser*'s measured characteristics show lower cache hits and higher misses than expected numbers. This 20-30% difference of cache behavior is reflected in the performance, since the performance of *analyser@cachebuster* is 0.39, which is 20% less than the anticipated performance of 0.5.

Table 4-2 Workload characteristics of *analyser@cachebuster*

domain	Dom1	Dom2
workload	*analyser*	*cachebuster*
cpuutil	0.50	0.50
cachehits(K)	15162.8	5243.3
cachemisses(K)	4970.7	9746.2
vmswitches	484.2	486.6
novmswitches	23.6	23.7
reads_issued	0.5	0.0
time_reading	1.4	0.0
writes_issued	0.4	0.0
time_writing	0.3	0.0
blocks	0.7	0.0

4.2.2. I/O Interference

Grep and *cat* have unique performance patterns compared with most of other applications. While they perform relatively well against CPU- and memory-intensive applications, their performance drops significantly against applications such as *ccrypt* and *gzip* (including themselves). Note that g*rep* and *cat* have high numbers for I/O characteristics. The applications that significantly affect the performance of *grep* and *cat* also have relatively high I/O characteristic numbers in Table 4-1. Our results indicate that there is a significant degree of performance interference between I/O-intensive applications. We show the workload characteristics of *cat@grep* in Table 4-3, to see how workload characteristics change depending on the background interference.

Table 4-3 Workload characteristics of *cat@grep*

domain	Dom1	Dom2
workload	*cat*	*grep*
cpuutil	0.06	0.01
cachehits(K)	3547.6	435.2
cachemisses(K)	197.1	60.0
vmswitches	252.0	154.0
novmswitches	0.0	0.0
reads_issued	51.2	48.9
time_reading	952.7	993.5
writes_issued	0.8	1.1
time_writing	16.3	24.3
blocks	149.3	151.1

Note that *cat*'s reads_issued numbers are significantly reduced (from 186.0 to 51.2), compared with the idle case. Meanwhile, the time_reading increased from 791.8 to 952.7. This indicates that *cat* is spending more time for fewer numbers of disk reading operations. *Cat* suffers from this inefficiency, resulting in a poor normalized score, 0.28. On the contrary, CPU utilization of *cat* becomes even smaller due to increased I/O waiting time. Therefore, CPU-intensive programs against *cat* are able to achieve better performance.

4.3. Application Clustering

With our findings about performance interference, we explore application clustering according to their measured characteristics. The application clusters would be useful for predicting performance of a new application since we can predict its performance by looking at performance of another application in the same cluster.

4.3.1. Clustering applications using performance scores and workload characteristics

For further analysis, we ran a hierarchical clustering algorithm using each application's performance score vector[3], which consists of normalized performance scores of an application against all the background applications. The resul-

[3] We used R (http://www.r-project.org), a free software tool for statistical computing, for our analysis and statistical modeling in section 4 and section 5.

tant dendrogram is presented in Figure 4-2. The height of the graph represents the distance between clusters; applications can be grouped into clusters based on the tree structure that comes from the analysis.

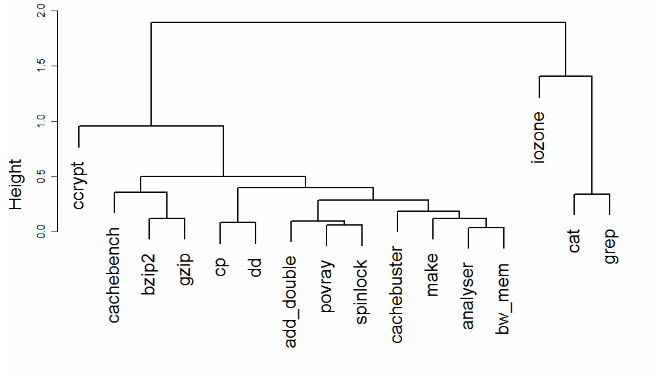

Figure 4-2 Workload clustering using scores

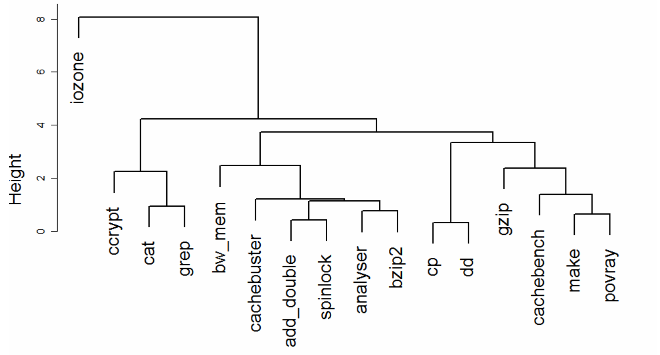

Figure 4-3 Workload clustering using system-level characteristics

The clustering dendrogram confirms the performance score graph shown earlier. *Cat* and *grep* have similar performance patterns; while *iozone* and *ccrypt* have their own performance curve, and all others follow a similar pattern.

In order to investigate the relationship between performance score and workload characteristics, we ran a hierarchical clustering algorithm with just our system-level characteristics. Because we have 10 characteristic variables, each application has a characteristic signature vector of length of 10 * 16 (the number of the background workloads). Since each workload characteristic has a different range, we scaled each variable v such that $0 \leq v \leq 1$.

Figure 4-3 shows the results for clustering using system level characteristics, which shows significant differences compared with Figure 4-2. We had expected to see similar clustering results, on the assumption that the shape of workload characteristics will have the most direct impact on the performance. One possible explanation for these results is that we gave all the characteristics equal weight. Since correlation of some workload characteristics with the performance is stronger than others, we decided to apply a weighted clustering algorithm.

4.3.2. Weighted Clustering

From Figure 4-2, we learned that large numbers of the applications, *add_double, analyser, bw_mem, bzip2, cp, dd, gzip, cachebench, cachebuster, make, povray*, and *spinlock*, have similar performance patterns. We propose that the characteristic variables changing less among those applications should have greater impact on the performance. In order to measure the rate of change, we calculate the standard deviation divided by the mean (coefficient of variation) for each characteristic variable.

Table 4-4 coefficient of variation for each characteristic variable for chosen applications

x	stddev(x)/mean(x)
cpuutil	0.28
cachehits(K)	0.96
cachemisses(K)	1.89
vmswitches	0.27
novmswitches	0.70
reads_issued	2.18
time_reading	3.53
writes_issued	2.28
time_writing	2.45
blocks	2.35

Table 4-4 shows the calculated values for each variable. From the table, we notice CPU utilization and virtual machine switching numbers stay relatively stable. This implies they are more important factors to determine the performance of applications. To reflect this, we take the inverse of a calculated value to get a weight for each variable. After multiplying weights by each corresponding column in our matrix, we get the following clustering dendrogram (Figure 4-4). We can see that weighted clustering gives us clustering results significantly closer to the performance score clusters.

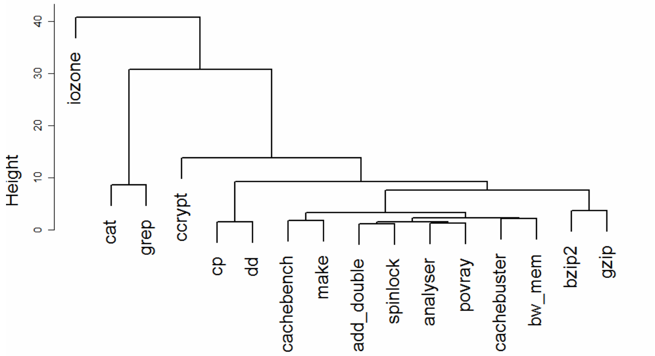

Figure 4-4 Weighted workload clustering with system-level characteristics

Because of this similarity, and its correspondence with intuition, we believe that weighting characteristics based on the variability of known workloads improves the clustering, and provides a basis from which to further research which characteristics have what significance for this purpose. Our future work will include isolating the particularly important characteristics, and determining which other characteristics our system (in its instrumented hypervisor and elsewhere in the platform) can produce.

5. APPLICATION PERFORMANCE PREDICTION

In section 4, we learned that certain types of applications can generate significant performance interference and showed application clusters according to them. While clustering helped us understand the relations between applications, however, clustering results alone were not enough for us to predict the performance score of new applications, since majority of applications fell into a single cluster. In this section, we describe our approaches to predict performance scores of unknown applications. These approaches derive normalized performance scores of an application from its measured system-level workload characteristics.

For evaluation, we randomly choose one application U from our experimental setup. By using the remainder of the applications as a benchmark set (B1, B2, …, Bn), we predict U's normalized scores against benchmark applications (i.e. NS(U@B1), NS(U@B2), … NS(U@Bn)), benchmark applications' normalized scores against U (i.e. NS(B1@U), NS(B2@U), … NS(Bn@U)), and U's normalized score against itself (NS(U@U)). Then, we compare predicted scores with actual measured scores.

5.1. Weighted Mean Method

Hoste et al. [11] predicted applications performance using program similarity of benchmark programs surrounding the application. We use a similar mechanism to predict performance of an unknown application U. A comparison of the two approaches will be discussed in a later section.

For similarity of two applications, we calculate distances of two foreground workload characteristic vectors. Since the high dimensionality of our data (10 system-level characteristics) can obscure the meaningful distances between data points [3], we use principal component analysis (PCA) [10]. PCA helps transfer our benchmark data points into more meaningful coordinates as well as reduce the number of dimensions of data. Once we transfer the benchmark space using PCA, we choose the most important principal components (PCs) that capture the most variance of the data. In the analysis described herein, we chose top four PCs, which account for around 85% of total variance. (Each represents 49%, 16%, 12%, and 8% of total variance, respectively.) We show factor loadings for the top four PCs in Figure 5-1.

To calculate the predicted score of U@Bn, we do as follows. First, in PCA-transferred space, we calculate Euclidean distances from the desired point, U@Bn, to all known benchmark results and choose the N closest data points as a near set. Similarity between the desired point and a point in the near set is defined as an inverse of distance. We, in turn, calculate the weight of each datum in a near set proportional to the similarity. Thus, we have

$w_i = s_i / \sum_N s_i$ where s_i is a similarity of an app i in the near set

Finally, we calculate a predicted score of U@Bn

NS(U@Bn) = $\sum_N w_i \cdot NS(i)$.

We show some of our prediction results Table 5-1. In these results, we choose *analyser* as an unknown application and N = 3. As a reference, we show the nearest data point to the prediction datum and the distance between them. For evaluation, we present mean, median, and maximum prediction error, where prediction error is calculated by | actual score – predicted score | ÷ actual score. Our prediction results for other applications will be presented later in this section.

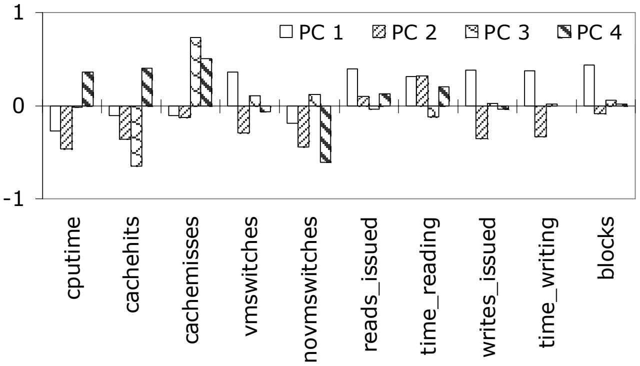

Figure 5-1 Factor loadings for top four PCs

Table 5-1 Sample prediction results for analyser using weighted mean method

Data points	Actual score	Predicted score	Prediction error	Nearest data point	Nearest distance
analyser@add_double	0.489	0.499	1.99%	bzip2@povray	0.0421
analyser@bzip2	0.450	0.455	1.18%	bzip2@cachebuster	0.0547
analyser@cachebuster	0.397	0.433	8.95%	bzip2@cachebuster	0.0628
analyser@cat	0.679	0.689	1.52%	povray@cat	0.1093
…					
analyser@make	0.461	0.455	1.45%	bzip2@make	0.0362
analyser@spinlock	0.484	0.477	1.44%	bzip2@make	0.0368
add_double@analyser	0.484	0.487	0.64%	add_double@cachebuster	0.0074
bzip2@analyser	0.464	0.473	2.01%	bzip2@bw_mem	0.0091
…					
make@analyser	0.435	0.429	1.27%	make@cp	0.0153
povray@analyser	0.465	0.469	0.91%	povray@cachebuster	0.0048
Average error			**1.81%**		
Median error			**1.25%**		
Max error			**8.95%**		

5.2. Linear Regression Analysis

5.2.1. Background

One of the most commonly used statistical procedures to model relationships between variables is regression analysis [7]. It relates a dependent variable Y with explanatory variables X_1, X_2, … X_n, used as predictors. A simple form of regression analysis is linear regression, in which we assume the dependent variable is a linear function of explanatory variables. Then we have,

$$\overline{Y} = a_0 + a_1 \cdot X_1 + a_2 \cdot X_2 + \ldots + a_n \cdot X_n$$

The goal of linear regression analysis is to find coefficients $a_0, a_1, \ldots, a_n$, to minimize error $|Y - \overline{Y}|$

5.2.2. Linear Regression Analysis on Score Prediction

We modeled our system using linear regression analysis. Normalized scores of unknown data points were a dependent variable and system-level workload characteristics were explanatory variables. A benchmark set of workload characteristics were used as training data to determine coefficients. Once coefficients are calculated using least squares method, we simply applied workload characteristic vectors to the equation to get predicted scores.

As sample results, we present regression coefficients and prediction results for *analyser* in Table 5-2 and Table 5-3.

Table 5-2 Linear regression coefficients for *analyser*

X	coefficient	X	coefficient
cputime	6.60E-01	reads_issued	5.70E-04
cachehits	-3.98E-10	time_reading	-4.41E-05
cachemisses	-6.62E-10	writes_issued	1.95E-05
vmswitches	-4.99E-04	time_writing	-7.13E-06
novmswitches	-1.06E-03	blocks	8.19E-04
a_0	4.33E-01		

Table 5-3 Sample prediction results for *analyser* using linear regression analysis

Data points	Actual score	Predicted score	Error
analyser@add_double	0.489	0.486	0.59%
analyser@bzip2	0.450	0.496	10.23%
analyser@cachebench	0.465	0.484	4.13%
analyser@cachebuster	0.397	0.487	22.72%
analyser@cat	0.679	0.685	0.95%
…			
analyser@make	0.461	0.476	3.07%
analyser@povray	0.479	0.484	1.14%
analyser@spinlock	0.484	0.478	1.34%
add_double@analyser	0.484	0.493	1.89%
bzip2@analyser	0.464	0.504	8.64%
…			
make@analyser	0.435	0.469	7.92%
povray@analyser	0.465	0.483	3.99%
spinlock@analyser	0.497	0.510	2.63%
analyser@analyser	0.480	0.493	2.69%
Average error			**7.3%**
Median error			**3.8%**
Max error			**38.5%**

5.3. Evaluation of Performance Prediction Methods

5.3.1. Performance Prediction for Various Workloads

We present performance prediction results for all applications we have in our experimental setup. Figure 5-2 and Figure 5-3 show median, mean, and maximum values of prediction errors for each workload with weighted mean method and regression analysis, respectively.

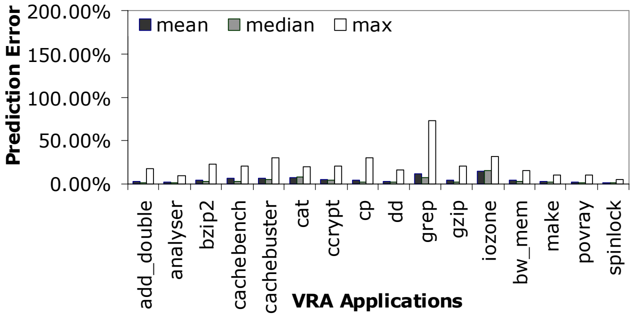

Figure 5-2 Mean, median, and max error of performance prediction with weighted mean method

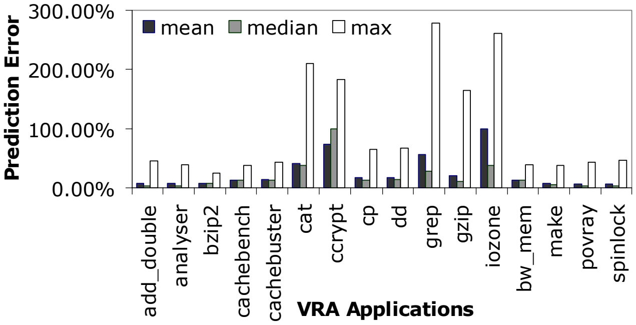

Figure 5-3 Mean, median, and max error of performance prediction with linear regression analysis

The weighted mean method predicts performance scores relatively correctly across applications. The application with the worst average prediction error is *iozone*, which has 14.3% average prediction error. Meanwhile, *grep* has the largest maximum prediction error. Average prediction error for *cachebuster* and *bw_mem* are 6.4% and 5.7%, respectively, which are higher than others. Median prediction error for most applications is below 6%, except *cat* and *grep*. Those applications, such as *cat, grep, cachebuster*, and *iozone*, are the ones that have high performance variation. We think the high variance made our algorithm harder to predict. Overall mean and median prediction error is 5.0% and 2.3%, respectively.

Linear regression analysis results are shown in Figure 5-3. Compared with weighed mean method, we can see that maximum prediction error for most applications is significantly high. Despite the maximum prediction error, the median prediction error is still low for some applications, such as, *add_double* and *povray*. One possible explanation for worse prediction results for linear regression analysis is that correlation between workload characteristics and performance is not linear. We leave non-linear regression analysis as future work.

5.3.2. Performance Prediction in Various Machines

We ran our experiments in a variety of physical hosts, in order to explore the applicability of our mechanisms across different physical hosts. We only show weighted mean method results, since that technique predicts more accurately than linear regression analysis in almost every case. Figure 5-4 is performance prediction results with a machine that has a bigger L2 data cache. Figure 5-5 shows results with a dual-processor SMP machine. Both results indicate that our performance prediction mechanisms work well with different hardware configurations. Overall mean and median prediction error in both machines are 7.9% and 2.5%, and 4.8% and 1.2%, respectively.

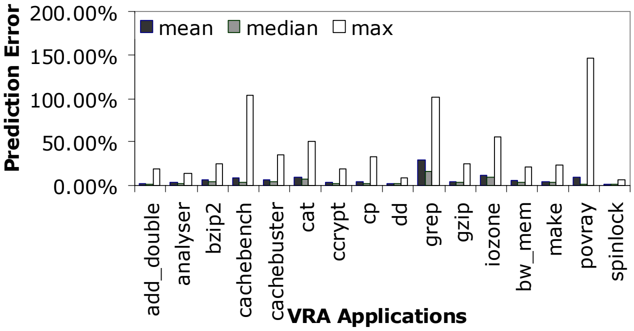

Figure 5-4 Mean, median, and max error of performance prediction in a machine of different cache size

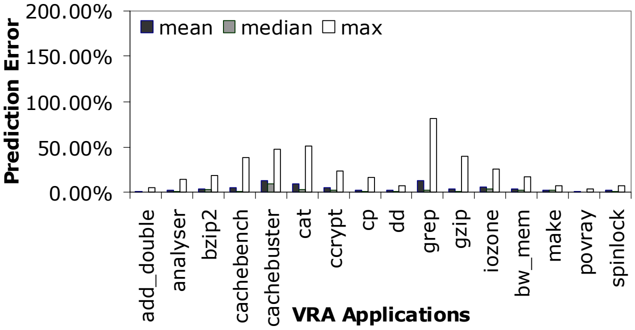

Figure 5-5 Mean, median, and max error of performance prediction in a dual processor machine

6. DISCUSSION

The performance prediction of applications will be especially beneficial for large/complex data centers, not only for improving relatively-static allocation of workloads to physical resources, but also as input into more complicated/dynamic orchestration systems. As trends toward virtualization continue, we assert that identification, prediction, and prevention of interference will improve performance and also boost overall utility (utilization).

We believe that one of the reasons our prediction results are good is that our benchmark programs cover a broad segment of the workload space. That is, a new application can be compared to considerably close application(s), from which we can predict the performance scores with reasonable accuracy. Thus, we believe that selecting a good set of benchmark programs is very important in real-world application of our techniques. The small margins of error in our predictions lend confidence that we will be able to use these techniques to further characterize workloads and predict interference; these data will be useful for a number of datacenter optimizations such as capacity planning and resource allocation among distributed applications.

With more than two concurrent VMs present, there is a chance that we may be able to predict performance scores of a VM using data collected with two VMs. This is because our prediction methods use workload characteristics solely from the domain, in which the target application is running. If otherwise, our methods will be required to collect different test data points for different numbers of VMs. We plan to extend our VRA system to support more than two VMs and research more details in the future.

7. RELATED WORK

Predicting the performance of applications using program characteristic similarity was discussed in [11][14]. They collected microarchitecture-independent variables to capture intrinsic behaviors of applications and predicted performance speed-up by calculating weighted average derived from program similarity in different machines. Our weighted mean method follows a similar approach to calculate the predicted score. However, our approaches differ in two aspects. First, we use system-level workload characteristics. Second, we consider the effects of performance interference. Previous work does not attempt to predict the performance score of competing applications.

Eeckhout et al. [6] studied the clustering of benchmark and input set pairs to find representative pairs in benchmark space. They used statistical data analysis techniques such as PCA to efficiently explore the workload space. Our work used similar techniques, but we focus specifically on program-program pairs rather than program-inputset pairs.

Resource contention between processes in a single OS is well-researched. Chandra et al. [4] predicted the L2 cache miss rates using three performance models for capturing the impact of cache sharing on co-scheduled threads. Settle et al. [15] introduced hardware activity vectors to monitor the access patterns on the cache, predicting inter-thread cache conflicts and improving job scheduling. In this paper, we successfully predicted overall system performance under both cache and I/O interference.

Other researchers have put their efforts into managing resources in a system to meet QoS requirements. Banga et al. [2] proposed new abstraction called a resource container. The resource container, which decouples resource management with process abstraction, is used for fair scheduling among activities. The Nemesis operating system [12] was designed to provide QoS guarantees to applications. Nemesis avoids QoS crosstalk by vertically-structured operating systems. However, these approaches require visibility to applications, which is not provided in virtualized environments due to isolation.

Performance interference among virtual machines is being researched. Gupta et al. [8] implemented XenMon to monitor the CPU usage of each guest and device driver domain and passed the usage information to a hypervisor scheduler for fair scheduling between applications that use device driver domains and ones that do not. Our system collects a greater variety of system characteristics including CPU usage for each domain. In our future work, we will explore the opportunity to use the collected system characteristics for hypervisor scheduling that leads to less performance interference among guest domains.

8. CONCLUSIONS AND FUTURE WORK

Virtualization is becoming widely used in large datacenters, due to the many advantages it brings. However, current technologies do not provide performance isolation, which can have significantly adverse effects on overall system performance. In this paper, we collected the system-level characteristics of different workloads collected in our experimental virtual environments and analyzed the collected data closely to characterize the workloads that generate intense performance interference. In addition to that, we developed performance prediction mechanisms for different combinations of workloads. Using our mechanisms, we were able to successfully predict the performance scores of the applications under performance interference with average error of approximately 5%.

We plan to extend our work in several directions. Foremost, our virtual resource allocation environment is being extended for more diverse workload scenarios – various numbers of concurrent VMs and different types of workloads such as network applications. We are also broadening our hardware diversity, so that we can determine the applicability of measurements (their "relative fitness"[13]) on one platform to another platform. We also are working to add instrumentation of more characteristics that might be beneficial in the grouping of applications by interference type and in improving our performance predictions. In the analysis realm, we plan to apply non-linear data analysis for performance score prediction, and to continue exploration of numerical techniques that will bring deeper revelation from the data our test environments generate.

9. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, and R. Neugebauer. Xen and the art of virtualization. In Proc. of the ACM SOSP. Oct. 2003.

[2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation, 1999

[3] S. Berchtold, C. Bohm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In Proc. of the PODS, pages 78-86, 1997

[4] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In Proc. of the HPCA. 2005.

[5] K. Duda and D. Cheriton. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a GeneralPurpose Scheduler. On Proceedings of the 17th Symposium on Operating Systems Principles, New York, 1999.

[6] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Workload design: selecting representative program-input pairs. In Proceedings of International Conference on Parallel Architectures and Compilation Techniques, Charlottesville, VA, 2002.

[7] J. Faraway. Practical Regression and Anova in R. July 2002.

[8] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforing Performance Isolation across Virtual Machines in Xen. In Proceedings of the 7th international Middleware Conference, Melbourne, Australia, 2006

[9] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Antfarm: Tracking Processes in a Virtual Machine Environment. In Proceedings of USENIX Annual Technical Conference, June 2006

[10] R. A. Johnson and D. W. Wichern. Applied Multivariate Statistical Analysis. Prentice Hall, fifth edition, 2002.

[11] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere. Performance prediction based on inherent program similarity. In Proceedings of the 15th international Conference on Parallel Architectures and Compilation Techniques, Seattle, Washington, USA, 2006.

[12] I.M. Leslie, D. McAuley, R. Black, T. Roscoe, P.T. Barham, D. Evers, R. Fairbairns, E. Hyden, The design and implementation of an operating system to support distributed multimedia applications. IEEE Journal of Selected Areas in Communications 14(7), 1996

[13] M. Mesnier, M. Wachs, G. Ganger, Modeling the Relative Fitness of Storage Devices, Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-05-106, 2005.

[14] A. Phansalkar and L. K. John. Performance prediction using program similarity. In Proceedings of the 2006 SPEC Benchmark Workshop, Jan. 2006.

[15] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, 2004.

[16] The Apache Software Foundation (http://www.apache.org)

[17] AIM Benchmark (http://sourceforge.net/projects/aimbench)

[18] Cachebench memory benchmark (http://icl.cs.utk.edu/projects/llcbench/cachebench.html)

[19] FreeBench (http://www.freebench.org/)

[20] IOzone Filesystem Benchmark (http://www.iozone.org)

[21] LMbench - Tools for Performance Analysis (http://www.bitmover.com/lmbench/)

[22] The Persistence of Vision Raytracer (http://www.povray.org)

[23] SPEC Benchmark suite (http://www.spec.org)

[24] VMware: Virtual Infrastructure Software (www.vmware.com)

[25] The Xen™ virtual machine monitor (http://www.cl.cam.ac.uk/research/srg/netos/xen/)