Virtual performance won't do: Capacity planning for virtual systems

Ethan Bolker[1], Yiping Ding

BMC Software

Abstract

The history of computing is a history of virtualization. Each increase in the number of abstraction layers separating the end user from the hardware makes life easier for the user but harder for the system capacity planner who must understand the relationship between logical and physical configurations to guarantee performance. In this paper we discuss possible architectures for virtual systems and show how naïve interpretations of traditional metrics like "utilization" may lead planners astray. Then we propose some simple generic prediction guidelines that can help planners manage those systems. We close with a benchmark study that reveals a part of the architecture of VMware[2].

[1] Ethan Bolker is also Professor of Computer Science at the University of Massachusetts, Boston.

[2] VMware is a registered trademark of VMware, an independent subsidiary of EMC.

1. Introduction

One can view the modern history of computing as a growing stack of layered abstractions built on a rudimentary hardware processor with a simple von Neumann architecture. The abstractions come in two flavors. Language abstraction replaced programming with zeroes and ones with, in turn, assembly language, FORTRAN and C, languages supporting object oriented design, and, today, powerful application generators that allow nonprogrammers to write code. At the same time, hardware abstraction improved perceived processor performance with microcode, RISC, pipelining, caching, multithreading and multiprocessors, and, today, grid computing, computation on demand and computing as a web service.

In order to master the increasing complexity of this hierarchy of abstraction, software and hardware engineers learned that it was best to enforce the isolation of the layers by insisting on access only through specified APIs. Each layer addressed its neighbors as black boxes.

But there's always a countervailing trend. Two themes characterize the ongoing struggle to break the abstraction barriers, and these are both themes particularly relevant at CMG.

First, the hunger for better performance always outpaces even the most dramatic improvements in hardware. Performance problems still exist (and people still come to CMG) despite the more than 2000-fold increase in raw processor speed (95% in latency reduction) and the more than 100-fold expansion in memory module bandwidth (77% in latency reduction) of the last 20 years [P04]. And one way to improve performance at one layer is to bypass the abstraction barriers in order to tweak lower levels in the hierarchy. A smart programmer may be smarter than a routine compiler, and so might be able to optimize better, or write some low level routines in assembler rather than in whatever higher level language he/she uses most of the time. A smart web application might be able to get better performance by addressing some network layer directly rather than through a long list of protocols.

Second, the economics of our industry calls for the quantification of performance – after all, that's what CMG is about. But layered abstractions conspire to make that quantification difficult. We can know how long a disk access takes, but it's hard to understand how long an I/O operation takes if the I/O subsystem presents an interface that masks possible caching and access to a LAN or SAN, or even the Internet. Measurement has always been difficult; now it's getting tricky too. And without consistent measurement, the value of prediction and hence capacity planning is limited.

That's a lot of pretty general philosophizing. Now for some specifics. One particular important abstraction is the idea of a virtual processor or, more generally, a virtual operating system.

Reasons for virtualization are well known and we won't go into detail here. Vendors provide it and customers use it in order to

- Isolate applications
- Centralize management
- Share resources
- Reduce TCO

In this paper we will present a framework which helps us begin to understand performance metrics for a variety of virtual systems.

2. A Simple Model for Virtualization

Figure 1 illustrates a typical computer system.

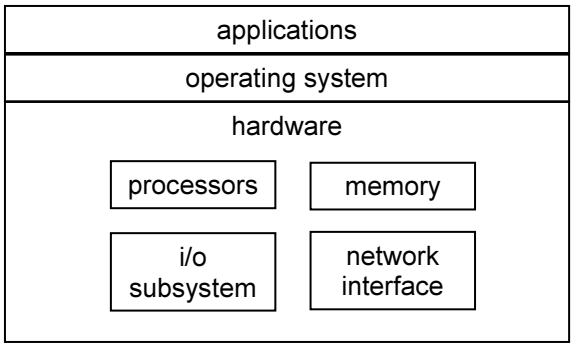

Figure 1. A basic computer system without virtualization.

In principle, any part of this diagram below the application layer can be virtualized. In practice, there are three basic architectures for virtualization, depending on where the virtualization layer appears. It may be

- below the OS (Figure 2)
- above the OS (Figure 3)

(or, possibly, in part above and in part below).

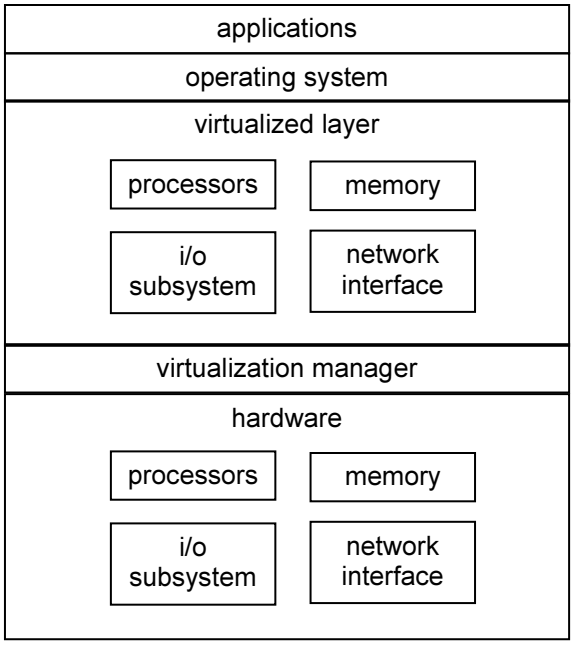

Figure 2. Virtualization layer below the operating system.

If the virtualization layer is below the operating system, then the OS has a very different view of the "hardware" available (Figure 2). If the virtualization layer is above the operating system (Figure 3), then the virtualization manager is, in fact, a new OS.

Historically, the first significant example of virtualization was IBM's introduction of VM and VM/CMS in '70s. Typical production shops ran multiple images of MVS[3] (or naked OS360). Various current flavors (each implementing some virtualization, from processor through complete OS) include

- Hyper-threaded processors
- VMware
- AIX micropartitions
- Solaris N1 containers
- PR/SM

Table 1 shows some of these virtualization products and where the virtualization layer appears.

	Vendor	**Below or Above OS?**
Hyper-threaded Processor	Intel	Below
VMware ESX Server	VMware (EMC)	Below
VMware GSX Server	VMware (EMC)	Above
Microsoft Virtual Machine Technology	Microsoft	Above
Micropartition	IBM	Below
Sun N1	SUN	Above and Below
nPar, vPar	HP	Below
PR/SM	IBM	Below

Table 1. Examples of virtualization products showing where the virtualization layer appears.

[3] Note for young folks – MVS has morphed into z/OS

In this paper we will focus on systems that offer virtual hardware to operating systems. Although we use VMware as an example in this paper, the model and methods discussed could be used for other virtualization architecture as well.

In the typical virtual environment we will study several **guest** virtual machines $G_1, G_2, \cdots, G_n$ running on a single system, along with a **manager** that deals with system wide matters. Each guest runs its own operating system and knows nothing of the existence of other guests. Only the manager knows all. Figure 3 illustrates the architecture. We assume that each virtual machine is instrumented to collect its own performance statistics, and that the manager also keeps track of the resource consumption of each virtual machine on the physical system. The challenge of managing the entire virtual system is to understand how these statistics are related.

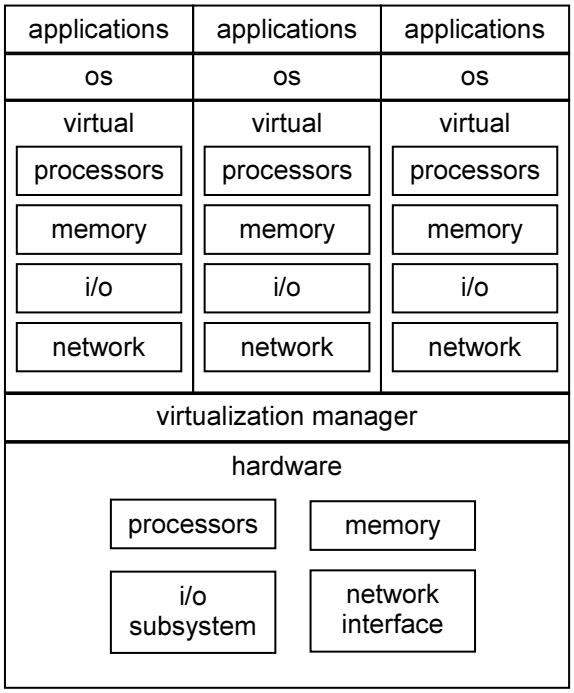

Figure 3. A virtualized system with 3 guests. Each guest has its own operating system, which may be different from the others. The Virtualization Manager schedules access to the real physical resources to support each guest.

3. Life before Virtualization

Perhaps the single metric most frequently mentioned in capacity planning studies is "processor utilization".

For a standalone single processor the processor utilization over an interval is the dimensionless number u defined as the ratio of the time the processor is executing "useful" instructions during the interval divided by the length of the interval. u is usually reported as a percentage rather than as a number between 0 and 1. The latter is better for modeling but the former is easier for people to process.

The usual way to measure the utilization is to look at the run queue periodically and report the fraction of samples for which it is not empty. The operating system may do that sampling for you and report the result, or may just give you a system call to query the run queue, in which case you do the sampling and the arithmetic yourself. The literature contains many papers that address these questions [MD02]. We won't do so here.

In this simple situation the utilization is a good single statistic for answering important performance questions. It tells you how much of the available processing power you are using, and so, in principle, how much more work you could get out of the system by using the remaining fraction 1- u.

That's true if the system is running batch jobs that arrive one after another for processing. The run queue is always either empty or contains just the job being served; when it's empty you could be getting more work done if you had work ready. But if transactions arrive independently and can appear simultaneously (like database queries or web page requests) and response time matters, the situation is more complex. You can't use all the idle cycles because transaction response time depends on

the length of the run queue, not just whether or not it is empty. The busier the system the longer the average run queue and hence the longer the average response time. The good news is that *often* the average queue length q can be computed from the utilization using the simple formula

$$q = \frac{u}{1-u}. \tag{3.1}$$

Now suppose the throughput is x jobs/second. Then Little's Law tells us that

$$r = q / x. \tag{3.2}$$

If each job requires an average of s seconds of CPU processing then $u = sx$ and we can rewrite formula (3.2) as

$$r = q / x = \frac{s}{1-u}. \tag{3.3}$$

The response time is greater than s because each job contends with others for processor cycles. The basic concepts presented above can be found in [B87] [LZGS]. We will use and interpret those basic formulas in the context of virtualization.

Measuring CPU consumption s in seconds is a historic but flawed idea, since its value for a particular task depends on the processor doing the work. A job that requires s seconds on one processor will take $t \times s$ seconds on a processor where, other things being equal, t is the ratio of the clock speeds or performance ratings of the two processors. But we can take advantage of the flaw to provide a useful interpretation for formula (3.3). It tells us that the response time R is the service time in seconds that this job would require on a processor slowed down by the factor $(1-u)$ – that is, one for which $t = 1/(1-u)$. So rather than thinking of the job as competing with other jobs on the real processor, we can imagine that it has its own slower processor all to itself. On that slower processor the job requires more time to get its work done. It's that idea that we plan to exploit when trying to understand virtualization.

We can now view the simple expression

$$u = x \times s$$

for the utilization in a new light. Traditionally, planners think of u both as how busy the system is and simultaneously as an indication of how much work the system is doing. We now see that the latter interpretation depends on the flawed meaning of s. We should use the throughput x, not the utilization u as a measure of the useful work the system does for us.

When there are multiple physical processors the system is more complex, but well understood. The operating system usually hides the CPU dispatching from the applications, so we can assume there is a single run queue with average length q. (Note that a single run queue for multiple processors is more efficient than multiple queues with one for each processor [D05].) Then (3.2) still gives the response time, assuming that each individual job is single threaded and cannot run simultaneously on several processors. Equation (3.1) must be modified, but the changes are well known. We won't go into them here.

When there is no virtualization, statistics like utilization and throughput are based on measurements of the state of the physical devices, whether collected by the OS or using APIs it provides. Absent errors in measurements or reporting, what we see is what was really happening in the hardware. The capacity planning process based on these

measurements is well understood. Virtualization, however, has made the process less straightforward. In the next section we will discuss some of the complications.

4. What does virtual utilization mean?

Suppose now that each **guest** runs its own copy of an operating system and records its own utilization V_i (virtual), throughput x_i and queue length q_i, in ignorance of the fact that it does not own its processors. Perhaps the **manager** is smart enough and kind enough to provide statistics too. If it does, we will use U_i to represent the real utilization of the physical processor attributed to guest G_i by the manager. We will write U_0 for the utilization due to the manager itself. U_0 is the cost or overhead of managing the virtual system. One hopes it is low; it can be as large as 15%.

5. Shares, Caps and Guarantees

When administering a virtual system one of the first tasks is to tell the manager how to allocate resources among the guests. There are several possibilities:

- Let each guest consume as much of the processing power as it wishes, subject of course to the restriction that the combined demand of the guests does not exceed what the system can supply.

- Assign each guest a share f_i of the processing power (normalize the shares so that they sum to 1, and think of them as fractions). Then interpret those shares as either **caps** or **guarantees**:

    - When shares are **caps** each guest owns its fraction of the processing power. If it needs that much it will get it, but it will never get more even if the other guests are idle. These may be the semantics of choice when your company sells fractions of its large web server to customers who have hired you to host their sites. Each customer gets what he or she pays for, but no more.

    - When shares are **guarantees**, each guest can have its fraction of the processing power when it has jobs on its run queue – but it can consume more than its share when it wants them at a time when some other guests are idle. This is how you might choose to divide cycles among administrative and development guests. Each would be guaranteed some cycles, but would be free to use more if they became available.

The actual tuning knobs in particular virtual system managers have different names, and much more complex semantics. To implement a generic performance management tool one must map and consolidate those non-standard terms. Here we content ourselves with explaining the basic concepts as a background for interpreting the meaning of those knobs with similar but different names from different vendors.

In each of these three scenarios, we want to understand the measurements reported by the guests. In particular, we want to rescue Formula (3.3) for predicting job response times.

6. Shares as Caps

The second of these configurations (shares as caps) is both the easiest to administer and the easiest to understand. Each guest is unaffected by activity in the other guests. The utilization and queue length it reports for itself are

reliable. The virtual utilization V_i accurately reflects the fraction of its available processing power guest G_i used, and the queue length q_i in Formula (3.3) correctly predicts job response time.

If the manager has collected statistics, we can check that the utilizations seen there are consistent with those measured in the guests. Since guest G_i sees only the fraction f_i of the real processors, we expect

$$V_i = \frac{U_i}{f_i}$$

As expected, this value approaches 1.0 as U_i approaches f_i .

Let S_i be the average service time of jobs in guest G_i, measured in seconds of processor time *on the native system*. Then $x_i \times S_i = U_i$. Since the job throughput is the same whether measured on the native system or in guest G_i, we can compute the average job service time on the virtual processor in guest G_i, s_i:

$$s_i = \frac{V_i}{x_i} = \frac{V_i}{\frac{U_i}{S_i}} = \left(\frac{Vi}{U_i}\right) S_i \qquad (6.1)$$

Thus V_i / U_i is the factor by which the virtual processing in guest G_i is slowed down from what we would see were it running native. That is no surprise. And there's a nice consequence. Although the virtual service time doesn't measure anything of intrinsic interest, it is nevertheless just the right number to use along with the measured virtual utilization V_i when computing the response time for jobs in guest G_i:

$$R_i = \frac{s_i}{1 - V_i} \qquad (6.2)$$

But, as we saw in the last section, you should not use either V_i nor s_i to think about how much work the system is doing. For that, use the throughput x_i. It's more meaningful both in computer terms and in business terms.

7. How contention affects performance – no shares assigned

In this configuration the planner's task is much more complex. It's impossible to know what is happening inside each guest using only the measurements known to that guest. Performance there will be good if the other guests are idle and will degrade when they are busy. Suppose we know the manager measurements U_i. Let

$$U = \sum_{i=0}^{n} U_i \qquad (7.1)$$

be the total native processor utilization seen by the manager. Note that the manager knows its own management overhead U_0.

In this complicated situation the effect of contention from other guests is already incorporated in the guest's measurement of its utilization. That's because jobs ready for service are on the guest's run queue both when the guest is using the real processor and when it's not. So, as in the previous section, we can use the usual queueing theory formulas for response time and queue length in each guest.

So far so good. But to complete the analysis and to answer what-if questions, we need to know how the stretch-out factor V_i / U_i depends on the utilizations U_j, $j = 0,1,\cdots i-1, i+1, \cdots, n$, of the other guest

machines. When the guest G_i wants to dispatch a job the virtualization manager sees the real system busy with probability

$$U - U_i = \sum_{\substack{j=0 \\ j \neq i}}^{n} U_j \,. \tag{7.3}$$

Note that the overhead U_0 is included in this sum. Thus the virtual system seen by G_i is slowed down by a factor $(1-(U-U_i))$ relative to the native system. So we expect to see

$$V_i = \frac{U_i}{1-(U-U_i)}\,. \tag{7.4}$$

In the analysis so far we have assumed that the manager could supply the true guest utilizations U_i. What if it can't – suppose we know only the measured values V_i, $i = 1,2,\cdots,n$? No problem (as they say). If we assume the virtualization management overhead U_0 is known we can think of the n equations (7.4) (one for each value of i) as n equations in the n unknowns U_i, $i = 1,2,\cdots,n$, and solve them. That reduces the what-if problem to a problem already solved. The solution is surprisingly easy and elegant. When we cross multiply to clear the denominators and reorganize the result we see that the equations (7.4) are actually linear in the unknown values U_i:

$$\begin{cases} U_1 + V_1 U_2 + \cdots + V_1 U_n = V_1(1-U_0) \\ V_2 U_1 + U_2 + \cdots + V_2 U_n = V_2(1-U_0) \\ \cdots \\ V_n U_1 + V_n U_2 + \cdots + U_n = V_n(1-U_0) \end{cases} \tag{7.5}$$

Solving by any standard mechanism (determinants, Cramer's rule) leads to

$$U_1 = \frac{(1-V_2)(1-V_3)\cdots(1-V_n)}{D} V_1(1-U_0) \tag{7.6}$$

where the denominator D is given by the formula

$$\begin{aligned} D = 1 &- \sum_{<i,j>} V_i \times V_j \\ &+ 2 \sum_{<i,j,k>} V_i \times V_j \times V_k \\ &- 3 \sum_{<i,j,k,l>} V_i \times V_j \times V_k \times V_l \\ &+ \\ &\cdots \\ &\pm (n-1) \times V_1 \times V_2 \times \cdots \times V_n \end{aligned} \tag{7.7}$$

To find the value of V_j, $j = 2,3,\cdots,n$ use equation (7.6) with 1 replaced by i in the obvious way.

Note that in any particular virtualization system it may or may not be possible to assign "no shares." Assigning equal shares may or may not have this effect.

8. How Contention Affects Performance – Shares as Guarantees

Although users often configure shares as caps, and may occasionally use no shares at all, it's clear that they will often want to use the tuning knobs provided by their particular system to provide shares as guarantees in order to achieve business goals that balance performance with the efficient use of computing resources.

When shares are guarantees, in a baseline system the guest utilizations V_i can be trusted to incorporate the stretch-out of U_i caused by contention from the other guests and can be used in normal ways to predict transaction response times. But formula (7.4) no longer

expresses the relationship between the guest's virtual utilization V_i and the manager's true utilizations U_i. Computing V_i from the U_j (or, using the equations in the previous section, the other guests' measured utilizations V_j) calls for a complex function

$$V_i = F(f_1,\cdots,f_n,U_1,\cdots,U_n),$$

which depends on the precise semantics of share assignments. In the several systems we have studied these are subtle and different, and often not encapsulated in a single parameter that can be normalized to our generic share f_i.

Nevertheless, we are still hopeful that we can find a reasonable generic approximation based on known analyses of priority queueing systems and fair share scheduling. One possible place to start is with the share algorithm developed for Solaris modeling [BD][BDR]..

9. VMware measurements

To test our models we ran a sequence of benchmarks on a virtual system with two guests. The manager was a VMware ESX Server; each guest ran Windows 2000. We instructed a load generator to force a specified utilization on each target machine by sending it a Poisson stream of computation intensive jobs (finding logarithms). The load generator was instrumented to collect statistics about the actual amount of work done and the job response time. While the benchmark was running we collected performance statistics for the guests and the VMware host using Collector and Agent from BMC Software.

We configured the load generators to run a sequence of experiments in which guest Bermuda was targeted to be busy 25% of the time while target utilization on guest Largo varied from 20% to 50%. CPU affinity in the dual processor VMware system was set so that the two guests competed with each other for the same processor but did not compete with the manager. Guests were told not to take advantage of Hyper-threading available on the physical system. Guests were assigned equal shares as guarantees; VMware does not provide a way to assign "no shares". Figures 4 and 5 show the results for Bermuda and Largo respectively. Utilizations are reported as percentages. Throughput and response time are essentially logarithms computed per second, scaled arbitrarily (but consistently) so that they can be displayed on the same graph. The former is an average over time. The latter captures the queueing delay when individual requests are backlogged by the randomness in their arrivals.

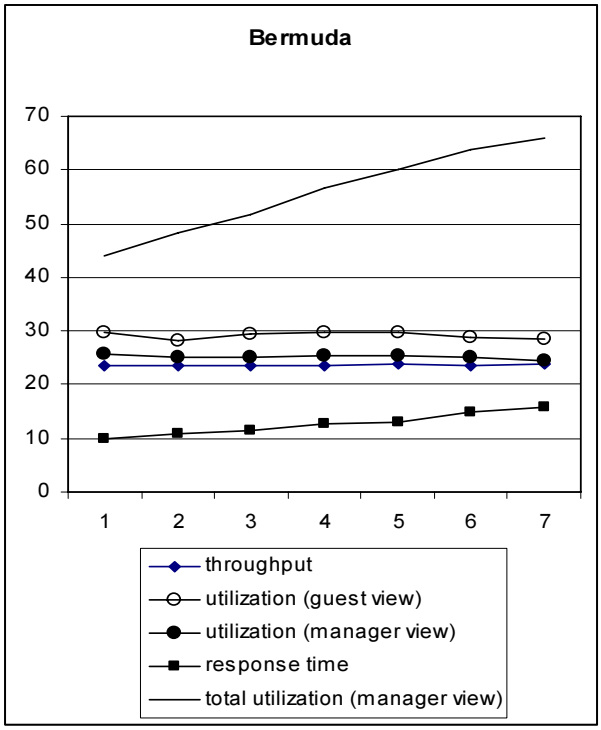

Figure 4. Experimental results from Bermuda, while Largo was working too.

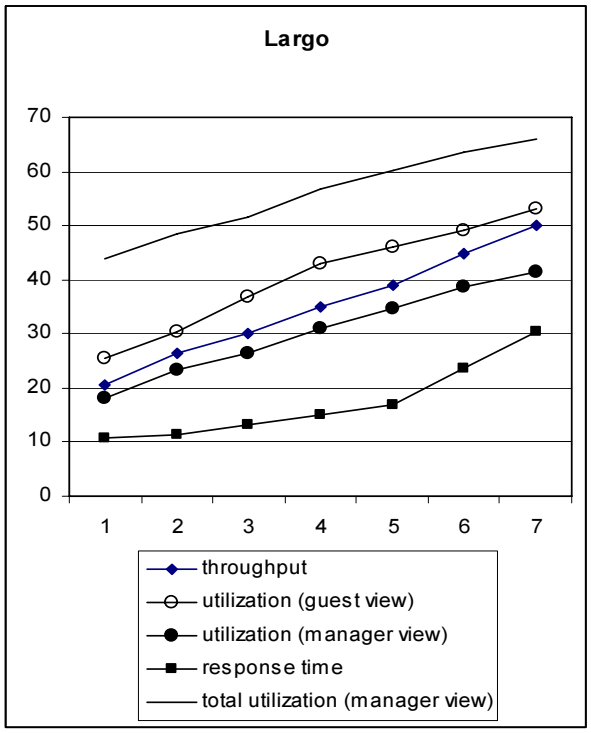

Figure 5. Experimental results from Largo, while Bermuda was working too.

Here is what we discover from those results.

- In all the experiments on both machines the guest's measurement of its utilization was larger than the utilization attributed to it by the manager. That we expected. But the amount of stretch-out does not vary significantly with the total load, as the analysis in Section 7 predicts, because that analysis is predicated on the assumption that no shares have been assigned. The proportional stretching is roughly constant for each machine, but different for the two machines. We do not understand why.

- The response time on each machine depends on the total utilization. That is particularly clear in the Bermuda data, where the load is constant but the response time increases from 10 to 15 seconds as the load on Largo increases. In the Largo data the effect is compounded by the increase due to the increase in load on Largo itself. When both guests ran at the same (approximate) load in Experiment 2, job response time was essentially the same on each. Had we not seen that happen we'd have worried about our benchmark driver.

10. Conclusions and Future Work

So far we have set down a generic framework that helps deal with the subtleties and complexities of measuring and modeling virtual systems. In particular, we have

- shown that processor utilizations measured by the guest and by the virtualization manager need not agree,

- discussed the relationship between those utilization measurements when no shares have been assigned,

- suggested the value of using throughput rather than utilization as the independent variable when attempting to answer what-if questions about transaction response time, and

- proposed a methodology for computing how activity in one guest can affect the performance in others.

In the future we hope to

- find a virtualization system that allows us to specify "no shares" so that we can validate the model in Section 7,

- continue our experiments on VMware and other systems in order to understand share allocation semantics, and

- develop a reasonably generic methodology for modeling at least the simplest of the share allocation semantics.

11. Acknowledgements

We would like to thank Ken Hu for valuable discussions on virtualization in general and VMware in particular and to Kangho Kim and Anatoliy Rikun for help in running experiments and analyzing the results.

12. References

[B87] Ethan Bolker, "A Capacity Planning/Queueing Theory Primer", *Proceedings of the 18th Computer Measurement Group Conference*, December 1987. (Best Elementary Tutorial Award)

[BDR] Ethan Bolker, Yiping Ding and Anatoliy Rikun, "Fair Share Modeling for Large Systems: Aggregation, Hierarchical Decomposition and Randomization", *Proceedings* of *the 30th Computer Measurement Group Conference* (December 2001), pp. 808-818

[BD] Ethan Bolker and Yiping Ding, "On the Performance Impact of Fair Share Scheduling", *Proceedings of the 30th Computer Measurement Group Conference*, (December, 2000), pp.71-8

[BB] Ethan Bolker and Jeff Buzen, "Goal Mode Scheduling", *Proceedings of the 28th Computer Measurement Group Conference*, December 1998.

[B85] Ethan Bolker, "Measuring and Modeling MVS under VM", *Proceedings of the 16th Computer Measurement Group conference*, December 1985.

[D05] Yiping Ding, "Bandwidth and Latency: Their Changing Impact on Performance," *Proceedings of the 35th Computer Measurement Group Conference*, December 2005.

[LZGS] E. Lazowska, J. Zahorjan, G. Graham, K. Sevcik, "Quantitative System Performance: Computer System Analysis Using Queueing Network Models," Prentice-Hall, 1984.

[MD02] Javier Munoz and Yiping Ding, "Sampling Issues in the Collection of Performance Data," *Proceedings of the 32nd Computer Measurement Group Conference*, December 2002.

[P04] David A. Patterson, "Latency Lags Bandwidth," Communications of the ACM, Vol. 47, No. 10, pp 71-75, October 2004