Database Systems on Virtual Machines: How Much do You Lose?

Umar Farooq Minhas
University of Waterloo

Jitendra Yadav
IIT Kanpur*

Ashraf Aboulnaga
University of Waterloo

Kenneth Salem
University of Waterloo

Abstract

Virtual machine technologies offer simple and practical mechanisms to address many manageability problems in database systems. For example, these technologies allow for server consolidation, easier deployment, and more flexible provisioning. Therefore, database systems are increasingly being run on virtual machines. This offers many opportunities for researchers in self-managing database systems, but it is also important to understand the cost of virtualization. In this paper, we present an experimental study of the overhead of running a database workload on a virtual machine. We show that the average overhead is less than 10%, and we present details of the different causes of this overhead. Our study shows that the manageability benefits of virtualization come at an acceptable cost.

1. Introduction

Virtual machine technologies are increasingly being used to improve the manageability of software systems, including database systems, and lower their total cost of ownership. These technologies add a flexible and programmable layer of software, the virtual machine monitor (VMM), between software systems and the computing resources (such as CPU and disk) that they use. The VMM allows users to define virtual machines (VMs) that have independent operating systems and can run different software, and it guarantees that these VMs will be isolated from each other: Faults in one VM will not affect other VMs, and the performance of a VM will not be affected by the activity in other VMs. The VMM also adds flexibility to the computing environment by providing capabilities such as dynamically changing the resource allocation to different VMs, suspending and resuming VMs, and migrating VMs among physical machines.

By running database systems in VMs and exploiting the flexibility provided by the VMM, we can address many of the provisioning and tuning problems faced by self-managing database systems. For example, we can allow many different database systems running in VMs to share the same physical machine while providing them with a guaranteed share of the resources of this machine [13]. This *server consolidation* is already widely used by many enterprises and service providers, since it reduces the number of servers required by an organization, which also reduces space, power, and cooling requirements. We could also use the VMM to change the resource allocation to a database system in response to fluctuations in the workload. As another example, we could use virtualization to simplify deploying database systems by providing VM images (i.e., suspended and saved VMs) with pre-installed and pre-configured database systems. In this case, deploying a database system is simply a matter of starting a VM from the saved VM image. Such pre-configured VM images are known as *virtual appliances* and are increasingly being used as a model for software deployment [16].

These are but a few examples of the benefits that virtual machine technologies can provide to database systems and other software systems. These benefits are widely recognized in the IT industry, which has led to users adopting virtual machine technologies at an increasing rate, and to hardware and software vendors providing new features aimed specifically at improving support for virtualization. However, the benefits of virtualization come at a cost since virtualization adds performance overhead. Our goal in this paper is to quantify this overhead for database systems. We ask the question: How much performance do we lose by running a database system in a virtual machine? What is the cost of the manageability benefits of virtualization?

To answer this question, we present a detailed experimental study of the performance of the TPC-H benchmark [15] on PostgreSQL. We compare the performance on Linux without virtualization to the performance in a VM using the Xen hypervisor [18], currently one of the most popular VMMs. We show that the average overhead is less than 10% and we report details on the nature and causes of this overhead. We view this as an encouraging result, since it means that the benefits of virtualization do not come at a high cost. Ours is not the first study to report the overhead of virtualization [1, 12]. However, to the best of our knowledge, we are the first to provide a detailed study of this overhead for database systems.

*Work done while the author was a summer intern at the University of Waterloo.

The rest of this paper is organized as follows. In Section 2 we present an overview of related work. Section 3 describes our experimental setup, and Section 4 reports our experimental results. Section 5 concludes.

2. Related Work

There has recently been an increasing interest in virtualization technologies [1, 17]. Virtualization provides flexibility and improved manageability by enhancing availability [3], performance isolation [5], security [4], memory management, and control of resources [11].

In this paper, we focus on the Xen VMM [18]. Xen was proposed in [1], where the authors provide a comparative performance evaluation against base Linux (with no virtualization), the VMWare VMM [17], and User Mode Linux. Using a variety of benchmarks, including database benchmarks, they show that the overhead of Xen is quite small compared to these other platforms. Their results are independently reproduced in [2].

In a more recent study, Padala et al. [12] provide an evaluation of the performance of the Xen and OpenVZ virtualization platforms for server consolidation. They use multi-tier applications that run on a web server and database server, and they report that the performance overhead for the database server is small for both virtualization platforms.

Gupta et al. [6] present XenMon, a performance monitoring tool for Xen. Using XenMon, they study the performance of a web server running in a Xen VM. In a similar study, Menon et al. [8] present Xenoprof, a toolkit to facilitate system-wide statistical profiling of a Xen system. The authors use Xenoprof to analyze the performance overhead of running a networking application on Xen.

These papers all study the performance overhead of Xen virtualization, but none of them provides a detailed study of this performance overhead for database systems.

3. Experimental Testbed

We use two identical machines for our experiments, one with Linux without virtualization and one with Xen virtualization. The machines are Sun Fire X4100 x64 servers, each with two 2.2GHz AMD Opteron Model 275 dual core processors, 8GB memory, and two 73GB SCSI 10K RPM drives. Our Linux system without virtualization runs SUSE Linux 10.1 with version 2.6.18 of the kernel and the ReiserFS file system. We refer to this system as the *Base* system.

Our Xen system uses Xen 3.1, the latest release of Xen at the time of this writing. Xen uses a special VM (a *domain* in Xen terminology) to control other VMs. This control domain is known as Dom0, and we allocate it 4GB of memory. We create a single VM (DomU) for running the database system, and we give it 3GB of memory and a single 10GB virtual disk mounted as a file in Dom0. Dom0 and DomU run the same version of Linux as the Base system, and their file systems are also ReiserFS. In addition to the operating system, the DomU file system contains PostgreSQL and the test database. Dom0 and DomU use one virtual CPU each, and these virtual CPUs are mapped to different physical CPUs on the machine, which ensures that Dom0 has enough resources to do its work without throttling DomU or competing with it for resources. We refer to this system in our experiments as the *Xen* system.

For our database system, we use PostgreSQL 8.1.3, which we refer to simply as *Postgres*. The workload we use for our experiments is the OSDL implementation of the TPC-H benchmark [10], with scale factor 1 (i.e., 1GB). We use the 22 queries of the benchmark, and not the update streams. This benchmark implementation is optimized for Postgres and utilizes several indexes to improve performance. The total size of the database on disk, including all tables and indexes, is 2GB. We set the Postgres buffer pool size (the `shared_buffer` parameter) to 2GB, ensuring that we can fit the entire database in memory in our warm buffer pool experiments. All the other Postgres configuration parameters are left at their default values. Postgres was identically configured for the Base and Xen systems. The Postgres client and server are both run on the same machine, and in the same domain for Xen (DomU). The client adds a negligible overhead to the machine, consuming well below 1% of the CPU and very little memory.

For our experiments, we run the 22 benchmark queries in identical settings on the Base and Xen systems, and we compare the performance of these two systems. Our performance measurements use Linux tools that are not Xen specific, so we use the same tools and measurement methodology for both systems. We conducted two different sets of experiments to measure the Xen overhead for different cases. In the *warm* experiments, the Linux file system cache and the Postgres buffer pool are warmed up before we do the measurement, and in the *cold* experiments we start with cold file system caches and buffer pool. For all our measurements, we repeat the experiment five times and report the average measurement obtained from these five runs. The variance in our measurements was very low in all cases.

4. Experimental Results

4.1. Warm Experiments

For these experiments, we run the 22 TPC-H queries once to warm up the buffer pool and file system cache, then we run them again and measure their performance. Table 1 presents the run time reported by Postgres for each query in the Base and Xen systems. The table also shows the absolute *slowdown*, defined as $(XenRunTime - BaseRunTime)$, and the relative slowdown, defined as $(XenRunTime - BaseRunTime)/BaseRunTime$.

	Base Runtime (secs)	Xen Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	14.19	15.30	1.11	7.82
Q2	0.12	0.17	0.05	40.39
Q3	5.20	6.98	1.78	34.35
Q4	0.74	1.07	0.33	44.00
Q5	4.53	5.99	1.46	32.21
Q6	1.40	2.12	0.73	52.03
Q7	4.09	5.32	1.23	30.14
Q8	1.39	1.98	0.59	42.05
Q9	10.99	12.81	1.81	16.49
Q10	5.04	6.36	1.32	26.17
Q11	0.78	0.94	0.16	20.82
Q12	1.85	2.73	0.88	47.32
Q13	14.02	15.27	1.25	8.93
Q14	0.66	0.90	0.24	37.12
Q15	1.24	1.66	0.42	34.32
Q16	1.89	2.18	0.29	15.17
Q17	0.39	0.47	0.08	19.45
Q18	9.38	11.54	2.17	23.12
Q19	5.26	6.33	1.07	20.41
Q20	0.59	0.94	0.35	60.03
Q21	2.79	3.65	0.86	31.03
Q22	1.59	1.70	0.10	6.58

Table 1. Overhead: Base vs. Xen.

The table shows that most queries experience a fairly large overhead when moving from the Base system to Xen, as indicated by the relative slowdown. We focus next on explaining this overhead. In the interest of space, we only present results for the ten queries whose run time in the Base system in Table 1 exceeds 2 seconds. The results for the other queries are similar, and can be found in [9].

In order to get an insight into the cause of the overhead, we use the `mpstat` tool to break down the overall run time of each query into user time and system time. Figure 1 shows the relative slowdown (defined similarly to Table 1) in user and system time for the ten selected queries.We can see that both the user time and the system time of almost all queries experience slowdown in Xen compared to Base. However, the slowdown in user time is very small compared to the slowdown in system time. This is expected since Xen adds overhead to system level operations and does not affect user level operations. User level code only experiences a minor slowdown that has been attributed in previous studies to the increased number of CPU cache misses under Xen [12], since the Xen code and data compete for CPU cache with user code and data. On the other hand, the slowdown in system time is quite significant (up to 154%) and can explain the run time overhead of Xen. So we focus next on the question: Where does the slowdown in system time come from? For these queries, system time is attributable to either system calls or page fault handling. We look into these two components next.

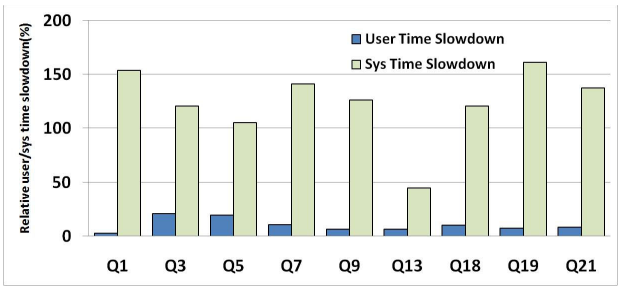

Figure 1. Slowdown: User vs. system time.

4.1.1 System Call Time

It is expected that system calls will be slower in the Xen system than the Base system. The way Xen is designed, many system calls have to go through the Xen hypervisor (which is why Xen virtualization is called *para-virtualization*). In the Base system, system calls are directly handled by the operating system. A longer system time can therefore be attributed to a longer execution path for system calls inside Xen. In this section, we are interested to know: (a) How much slower are system calls in Xen?, and (b) How much of the overhead of virtualization can be attributed to slowdown in system calls?

With a focus on these questions, we use the `strace` tool to collect the number of system calls made by each query (more precisely, by the Postgres process executing the query) and the time to serve these system calls. Table 2 presents these results for the ten selected TPC-H queries in the Base and Xen systems. The table also presents the relative slowdown in serving system calls, defined as $(XenSysCallTime - BaseSysCallTime)/BaseSysCallTime$.

As expected, the total time to serve system calls is higher inside Xen, increasing up to 871% compared to the Base system (except for Q13). Also as expected, the number of system calls in the Base and Xen systems is very similar since the program behavior of Postgres does not change when it is run in a VM. The `strace` tool consistently reports a small difference between Base and Xen of around 15 system calls. Most importantly, we note that for all queries, system call time is a minor fraction of the total system time. Thus, while system calls in Xen are significantly slower than the Base system, this is not a major cause of slowdown for database queries, since these queries do not spend that much time on system calls. We next look into the second component of system time: time to handle page faults.

4.1.2 Page Fault Handling Time

A page fault is an exception generated by hardware when the memory page accessed by the current instruction has not been loaded into physical memory (a *major page fault*), or has been loaded into memory for some other process but

	Base			Xen			Rel
	Number of SysCalls	SysCall Time (ms)	System Time (ms)	Number of SysCalls	SysCall Time (ms)	System Time (ms)	SysCall Time SlwDwn (%)
Q1	112	0.02	390.34	95	0.03	989.80	82.05
Q3	226	0.40	568.66	209	3.92	1253.02	871.44
Q5	204	0.13	557.38	188	0.42	1144.08	225.51
Q7	221	0.92	536.52	205	5.47	1294.16	491.20
Q9	348943	4.19	782.34	348926	11.45	1768.40	173.35
Q10	224	1.34	542.92	206	6.09	1219.52	353.00
Q13	34974	59.92	751.00	34957	34.92	1085.00	-41.73
Q18	308	16.97	962.12	291	65.89	2123.12	288.34
Q19	149	0.91	392.10	132	2.87	1023.72	215.08
Q21	579	0.02	451.40	562	0.05	1072.00	203.45

Table 2. System Call Time.

	Base Page Faults	Xen Page Faults	Page Faults per Second	Rel SlwDwn (%)
Q21	272679	272655	74700	31.03
Q7	351018	350303	65817	30.14
Q5	343477	342106	57136	32.21
Q10	344395	340994	53593	26.17
Q3	370884	368685	52814	34.35
Q19	288348	286636	45261	20.41
Q18	476445	473605	41029	23.12
Q9	364944	362777	28326	16.49
Q1	270281	269217	17594	7.82
Q13	98128	97738	6400	8.93

Table 3. Page Fault Handling: Base vs. Xen.

is not mapped to the address space of the faulting process (a *minor page fault*). Page fault handling is a significant source of complexity for VMMs, including the Xen hypervisor, so it is important to study their contribution to the observed overhead. To do so, we measure the number of page faults generated by each query, and we attempt to establish a relationship between slowdown and page faults.

In our experiments we only observe minor page faults. Our settings are chosen such that all pages required by the queries can fit into physical memory and are loaded in the warmup phase, thus avoiding major page faults. Minor page faults, on the other hand, can arise due to sharing of code and data pages between processes.

We use the `sar` tool to measure the number of page faults by each query, and we report the results in Table 3. The table also shows the relative slowdown of the queries (from Table 1). The number of page faults by each query shows a slight variation between Base and Xen but this is not significant. The important observation is that there is a strong correlation between relative slowdown and page faults per second, which is also shown in the table and is defined as the number of page faults generated in Xen divided by the total run time of the query in Xen. In general, if a query has a higher number of page faults per second, it will incur a higher run time overhead in Xen. To highlight this correlation, Table 3 is sorted by the number of page faults per second. This important conclusion establishes the fact that page faults are a major cause of database system slowdown in Xen.

We investigated the reason behind the page faults and found that the majority of them are caused by accesses to database pages in the shared buffer cache of Postgres. Like many database systems, Postgres uses worker processes, known as `postmaster` processes, to execute user queries, and they all use one shared buffer cache for database pages. When a `postmaster` process accesses a database page (from a table or index), this page may already be in the shared buffer cache, but the memory pages in which this database page resides may not be mapped to the address space of the process. The process needs to map these pages to its address space, which causes minor page faults.

This discussion implies that it takes longer to handle a page fault in Xen than in the Base system. To verify this, we measure the time to handle a page fault in the Base and Xen systems using the `lmbench` tool kit [7]. The results indicate that it takes 1.67μs and 3.5μs to handle a page fault in the Base and Xen systems, respectively. This direct measurement shows that page faults in Xen are more than twice as expensive as they are in the Base system.

Since the *page fault rate* (page faults per second) is strongly correlated to overhead, page fault handling is likely the major source of overhead for Xen. We turn our attention next to reducing this overhead.

4.1.3 Reducing Page Fault Overhead

To reduce page fault overhead, we attempt to reduce the number of minor page faults that happen when a `postmaster` process maps pages from the shared buffer cache to its own process space. The key to reducing these page faults is to realize that they only happen *the first time the process touches a memory page from the shared buffer cache*. Once a page is mapped to the address space of a `postmaster` process, the process can reuse this page without faulting.

	Base Runtime (secs)	Xen Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	13.3	14.04	0.74	5.55
Q3	4.61	5.82	1.21	26.23
Q5	4.14	4.97	0.84	20.22
Q7	3.52	3.66	0.14	3.91
Q9	10.52	11.36	0.83	7.91
Q10	4.57	4.69	0.12	2.58
Q13	13.36	14.1	0.75	5.59
Q18	8.86	10.13	1.27	14.36
Q19	4.84	5.05	0.22	4.46
Q21	2.3	2.48	0.18	7.84

Table 4. Overhead for Single Connection.

In the previous experiments, each query is individually run from the command line using the `pgsql` Postgres client, which means that there is a different client process and a different database connection for each query. In the Postgres architecture, whenever a new connection to the database is initiated, the master server process spawns a child worker process (a `postmaster` process) that is responsible for handling requests on the new connection. This means that in our case each query runs in a new `postmaster` process, which needs to map the buffer pool pages it uses to its address space. Each query thus causes a large number of minor page faults. To reduce the number of page faults, we repeat the experiments in Section 4.1, but we run all queries in one `pgsql` client, using one database connection and one `postmaster` process.

Table 4 presents the overhead in this case for the ten selected queries. In this new setting, the overall run time of all queries decreases in both Base and Xen. More importantly, the absolute and relative slowdown values are also lower compared to Table 1. The decrease in run time is due to a significant decrease in system time, which in turn is due to a reduction in the number of page faults. Other components of run time are not significantly affected.

To verify that the number of page faults is indeed reduced, we conduct an experiment in which we establish a database connection and run the same query multiple times on this connection, measuring the number of page faults in each run. Figure 2 presents the results for the ten longest running TPC-H queries. The number of page faults in the second and later runs of each query is almost identical, and is a lot smaller than the number of page faults in the first run. After the first run, the required pages from the shared buffer cache are mapped to the address space of the `postmaster` process and can be used without faulting.

Using one database connection for all queries is a fairly simple way to reduce the overhead of virtualization. Indeed, many applications do use one connection for all their queries, so these experiments do not suggest a new way of writing database applications. However, they do show that

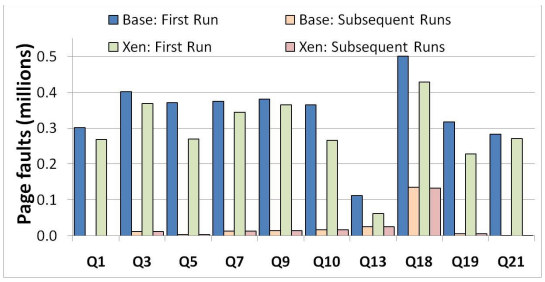

Figure 2. Page Faults for Single Connection.

virtualization introduces new types of overheads, and that there may be simple ways to reduce these overheads.

The average relative slowdown of the 22 TPC-H queries using one database connection is only 9.8%. This can be further reduced by using newer server CPUs that have hardware support for virtualization and include features that enable, for example, faster page fault handling. The CPUs of the machines that we use in our experiments do not have this support, so the 9.8% can be viewed as a worst case performance overhead that can likely be improved.

4.2. Cold Experiments

Next, we turn our attention to the case where the queries run on a cold database. For this set of experiments, we restart Postgres and flush the Linux file system caches in both DomU (the Postgres VM) and Dom0 (the control VM) before running each query. This ensures that all data required by a query is read from disk, thereby bringing an important (and usually very costly) factor into play, namely physical disk I/O. It is generally accepted that Xen adds a high overhead to the I/O path. In Xen, every I/O request in DomU is forwarded to Dom0 where it is serviced. Data that is read (or written) by Dom0 is copied to (or from) DomU. This is a complex process, so we expect to see larger slowdowns in this experiment.

We perform cold runs of the 22 TPC-H queries in the Base and Xen systems. Table 5 reports the run time of each query in both cases and the slowdown between Base and Xen. Surprisingly, the slowdown numbers are not high. The I/O path in Xen is not as slow as commonly believed, and the cost of virtualization is low even in the cold case.

We obtain the run time break down of these queries as in the warm case. In these runs, system call and page fault numbers are unchanged from the warm case, but they represent a small fraction of query run time. As expected, most of the run time of these queries is spent in the `iowait` state, waiting for disk I/O. On average, the queries spend 77% and 79% of their run time in the Base and Xen systems, respectively, waiting for disk I/O. In the warm case, query run time did not have an `iowait` component.

What is surprising and rather counter-intuitive in the results is that some queries run *faster* in Xen than in the Base

	Base Runtime (secs)	Xen Runtime (secs)	Abs SlwDwn (secs)	Rel SlwDwn (%)
Q1	22.12	22.09	-0.04	-0.17
Q2	2.14	2.25	0.11	5.17
Q3	26.48	29.88	3.39	12.81
Q4	59.62	46.07	-13.55	-22.73
Q5	24.24	27.89	3.66	15.08
Q6	19.86	22.57	2.71	13.62
Q7	25.28	28.89	3.61	14.28
Q8	171.19	178.42	7.23	4.22
Q9	798.9	776.21	-22.69	-2.84
Q10	24.00	28.14	4.14	17.25
Q11	3.11	3.81	0.70	22.35
Q12	51.46	43.92	-7.54	-14.65
Q13	17.10	18.01	0.91	5.35
Q14	20.18	24.06	3.89	19.27
Q15	20.09	22.94	2.85	14.17
Q16	6.94	7.83	0.89	12.81
Q17	29.45	31.83	2.38	8.08
Q18	26.88	31.46	4.58	17.03
Q19	20.67	23.13	2.45	11.87
Q20	277.16	280.71	3.56	1.28
Q21	623.30	612.61	-10.69	-1.71
Q22	26.00	22.03	-3.97	-15.26

Table 5. Overhead for Cold Runs.

system. In our final experiment, we try to explain why this interesting effect happens. We use the `iostat` tool to measure the amount of data read from the database disk by each query. For the Base system, we only need to monitor the physical disk storing the database. However, for the Xen system we need to monitor both the physical disk accessed by Dom0 and the virtual disk inside DomU.

Table 6 presents the amount of data read in each case and the `iowait` time in the Base system and in DomU. It is clear that for the queries that run faster inside Xen, the I/O wait time is less than the Base system. The speedup of these queries can be entirely attributed to this reduction in I/O wait time. Fortunately, this reduction can be explained easily by looking at the amount of data read by each query. The Base system and DomU read almost the same amount of data. This is expected since Postgres is accessing the same data in both cases. However, if we look at the amount of data read in Dom0, we find that for the faster queries, Dom0 reads a lot more data than required by DomU.

For these queries, Dom0 is aggressively prefetching data based on the disk access pattern of DomU. This causes more data than necessary to be read from disk, but it helps performance by removing I/O from the critical path of query execution and hence reducing I/O wait times. To explain the cause of this prefetching, we note that the Dom0 file system sees the DomU disk as one large file. Reads in DomU due to Postgres read requests and prefetching requests from the DomU file system are all translated to reads on this one file in Dom0. This causes the file system in Dom0 to see many reads on this file, and to prefetch aggressively from it. In particular, prefetching requests made by the DomU file system cause larger amounts of data to be prefetched in Dom0. If we turn the Linux prefetching mechanism off in DomU, the amount of data read is very close to the actual data required by the queries and the queries no longer run faster inside Xen. More details are available in [9].

The point of this experiment is not to suggest using virtualization to implement prefetching. Instead, the experiment is meant to illustrate that virtualization does not cost much even for the cold case. The queries that are slower in Xen experience an average slowdown of 12.2%, and those that are faster in Xen experience an average "slowdown" of -9.6%. Overall, the average slowdown for all 22 TPC-H queries is 6.2%. As in the warm case, these numbers are for a simple system that is not highly optimized. Optimizations such as using raw disk in Dom0 for the DomU virtual disk can improve performance, so this overhead can be viewed as a worst case overhead that can likely be further improved.

5. Conclusion

There are many advantages to running a database system in a virtual machine. In this paper, we demonstrate that these advantages do not come at a high cost in performance. Using a TPC-H workload running on Postgres in a Xen virtual machine environment, we show that Xen does indeed introduce overhead for system calls, page fault handling, and disk I/O. However, these overheads do not translate to a high overhead in query execution time. System calls and page faults represent a minor fraction of the run time of a query. Disk I/O is a significant fraction of run time in the cold case, but it is not slowed down by much, and it can sometimes be taken off the critical path. Our hope is that these findings will encourage further research in the area of virtualization and self-managing database systems.

Possible directions for future work include measuring the performance overhead of virtualization for other database workloads with different applications and resource demands. For example, we could use TPC-C [14] or other OLTP workloads. Since OLTP workloads have different characteristics (e.g., more updates), the results may be different. Additionally, we could evaluate different virtualization environments such as OpenVZ, VMWare, and User Mode Linux specifically for database workloads. We could also evaluate different database systems such as MySQL, Oracle, and DB2. A very interesting direction for future work is to conduct a study with multiple workloads running concurrently on different VMs sharing the same physical machine. This would allow us to investigate the effect of interference among workloads on performance, and to study how well the VMM isolates the performance of the different workloads from each other.

	Base		Xen		
	Data Read (MB)	**IOWait (secs)**	**Dom0 Data (MB)**	**DomU Data (MB)**	**IOWait (secs)**
Q1	1040	6.27	1045	1040	6.23
Q2	41	1.93	43	41	2.04
Q3	1386	17.73	1392	1385	23.05
Q4	**920**	**56.16**	**1347**	**920**	**44.6**
Q5	1311	16.63	1318	1311	18.13
Q6	1056	15.87	1062	1056	20.58
Q7	1342	17.46	1349	1342	23.58
Q8	623	166.99	955	623	174.34
Q9	**1512**	**778.41**	**2660**	**1511**	**759.66**
Q10	1308	16.2	1315	1308	17.93
Q11	145	2.15	147	145	2.96
Q12	**1302**	**46.53**	**1325**	**1302**	**40.24**
Q13	269	3.60	271	269	1.79
Q14	539	17.81	998	539	23.00
Q15	871	16.68	1047	871	21.09
Q16	205	4.80	207	205	6.36
Q17	90	28.71	162	90	31.15
Q18	1308	15.19	1315	1308	17.58
Q19	1082	13.39	1088	1082	13.22
Q20	541	273.22	1034	542	277.45
Q21	**1127**	**613.49**	**2175**	**1129**	**604.99**
Q22	**106**	**24.18**	**124**	**106**	**20.55**

Table 6. Disk Activity and I/O Wait: Base vs. Xen.

References

[1] P. T. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. ACM Symp. on Operating Systems Principles (SOSP)*, 2003.

[2] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews. Xen and the art of repeated research. In *Proc. USENIX'04, FREENIX Track*, June 2004.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. ACM/USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, May 2005.

[4] R. Figueiredo, P. A. Dinda, and J. Fortes. Resource virtualization renaissance. *IEEE Computer*, 38(1), May 2005.

[5] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in Xen. In *Proc. ACM/IFIP/USENIX 7th International Middleware Conf.*, November 2006.

[6] D. Gupta, R. Gardner, and L. Cherkasova. XenMon: Qos monitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs, 2005.

[7] LMbench: Tools for Performance Analysis. http://lmbench.sourceforge.net/.

[8] A. Menon, J. R. Santos, Y. Turner, G. J. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. Virtual Execution Environments (VEE'05)*, June 2005.

[9] U. F. Minhas. A performance evaluation of database systems on virtual machines. Master's thesis, University of Waterloo, 2007. Also available as University of Waterloo Computer Science Technical Report CS-2008-01, January 2008.

[10] OSDL Database Test Suite 3. http://sourceforge.net/projects/osdldbt.

[11] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc. EuroSys Conf.*, March 2007.

[12] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. G. Shin. Performance evaluation of virtualization technologies for server consolidation. Technical Report HPL-2007-59, HP Labs, April 2007.

[13] A. A. Soror, A. Aboulnaga, and K. Salem. Database virtualization: A new frontier for database tuning and physical design. In *Proc. Workshop on Self-Managing Database Systems (SMDB'07)*, April 2007.

[14] TPC-C: An On-line Transaction Processing Benchmark. http://www.tpc.org/tpcc/.

[15] TPC-H: An Ad-hoc, Decision Support Benchmark. http://www.tpc.org/tpch/.

[16] Virtual Appliances. http://www.virtualappliances.net/.

[17] VMware. http://www.vmware.com/.

[18] XenSource. http://www.xensource.com/.